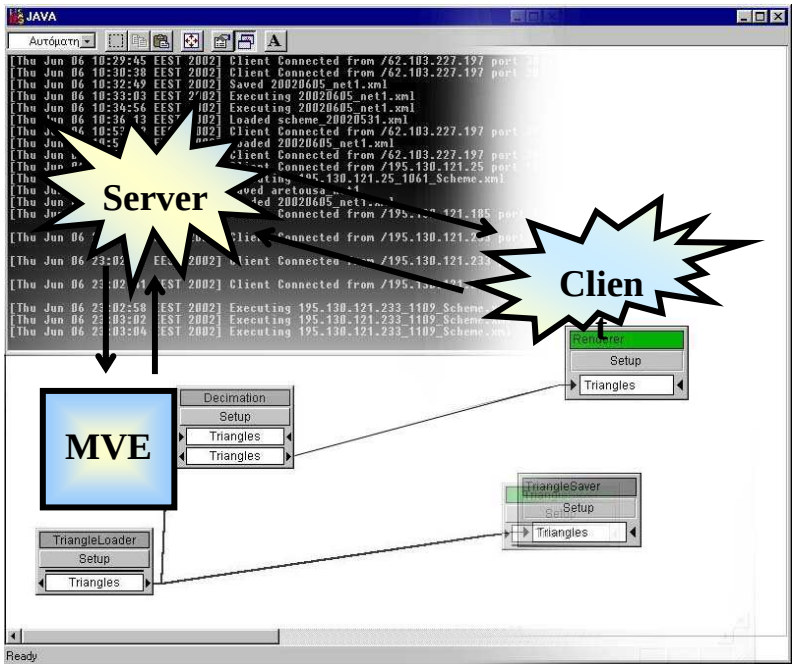
University of Ioannina

Department of Computer Science

MSc Thesis of **Giannis Kyriazis** kyriazis@cs.uoi.gr

A Client - Server System for Remote Internet Access to a Modular Visualization Environment



Advisor: **Ioannis Fudos** fudos@cs.uoi.gr

Ioannina, June 2002

Acknowledgements

I dedicated all my efforts to this thesis during the second year of my post-graduate studies. Nevertheless, I would have never completed the research and implementation of this project without the help and support of the people I am associated with. First of all, I would like to thank my advisor, professor Ioannis Fudos (fudos@cs.uoi.gr), for sharing his knowledge with me and supporting me during the course of our study. I would also like to thank our associates at the University of West Bohemia in the Czech Republic, for providing us with the Visualization Environment on which our implementation was based. Martin Franc (marty@kiv.zcu.cz) and Michal Rousal (rousal@kiv.zcu.cz) designed and implemented the Modular Visualization Environment as part of their PhD degree, under the supervision of professor Vaclav Skala (skala@kiv.zcu.cz). For their contribution, I owe them my gratitude.

Giannis Kyriazis

Abstract

In this project we present an approach to collaborative 3D Graphics processing and visualization through a Client – Server System that is used to access a Modular Visualization Environment (MVE) remotely through the Internet. The purpose of this study is to provide an efficient protocol for the communication between the client and the server, and between the server and the environment.

The Visualization Environment, which is the back-end of our system, is a modular environment. This means that its operation is based on a set of modules, which are individual components, and are responsible for various functions that are applicable to a representation of a 3D graphics object. Such functions can be:

(1) loading an object from a file or a stream,
(2) modifying the object,
(3) saving the object to a file, and
(4) rendering the resulting object.

These modules are designed in a common form, and each module has a set of parameters, inputs and outputs. The parameters define the behavior of the module, and they depend on the task the module is responsible for. The inputs of the module are arguments that are necessary for the module to run. An input to a module is usually an object that resulted of another module's execution. The outputs of a module are the resulting objects that the module has produced with its execution. Several modules connected properly with each other can form a scheme, which may perform complex computations to the objects it processes. This scheme can be executed, or it can be saved so that we can carry on modifying it in the future.

The server, which acts as a middleware between the MVE and the client, is responsible for providing the client with the data required to operate and offer its services to the remote user. The server acquires the information it needs from the MVE.

The client is the front-end of the system, and it is the part of the system, which the end user gets to interact with. The client downloads the list of available modules

on the MVE, by making a proper request to the server. Then the user can develop his scheme by inserting modules into it, setting their parameters and connecting them with each other. When the user has created a complete scheme, he may execute it and get the results. If the results contain objects for visualization, the client loads these objects and renders them. The rest of the results, which may be objects that are to be saved in a file, are kept on the server's computer, since the client does not need to posses this data. Instead of executing his scheme, the user may also save it for further processing. The scheme is saved on the server's computer, so that the user may retrieve it the next time he connects to the server, and more important, so that other users may use this scheme and maybe modify it. The client offers this feature, of saving and loading schemes, and it is achieved by making proper requests to the server. The MVE is not involved at this point.

The protocol for the communication between the client and the server, and the server and the MVE, defines the form of the data that is transmitted, the content of the data, and the moments when data needs to be transmitted. The communication is ordered as follows:

- When the client connects it requests the list of currently available modules
- The server calls MVE to get the list and sends it to the client
- When the client has created a scheme he can send it to the server for execution
- The server calls MVE to execute the scheme, and sends the results to the client
- Instead of executing the scheme, the client may send it to the server for saving
- The client may also load a scheme. It requests the list of available schemes and when the server sends this list it chooses a scheme. The server sends this scheme.

The data is always in XML format and contains the absolute information needed for the whole system to operate. Some of the parameters of the modules in the MVE have been omitted, and default values have been set in their place, so that we can reduce the complexity and the amount of the data to be transmitted.

The performance of our system has been more than satisfying. The time it takes to complete the described tasks was satisfactory, and the status of the network or the computers involved does not seem to influence the overall performance. The system performed well, even when the network was loaded with traffic, or the related computers are low on resources. In conclusion, we have designed and implemented a

system that provides remote Internet access to the MVE, with the minimum communication possible, that performs well, in various states of the involving resources.

Περίληψη

Στην εργασία αυτή παρουσιάζουμε μια προσέγγιση για collaborative επεξεργασία και απεικόνιση τρισδιάστατων αντικείμενων γραφικών με τη βοήθεια ενός συστήματος εξυπηρέτη – πελάτη που χρησιμεύει στην απομακρυσμένη προσπέλαση από το Internet σε ένα modular περιβάλλον οπτικοποίησης. Στόχος της έρευνας αυτής είναι να παρέχουμε ένα αποδοτικό πρωτόκολλο για την επικοινωνία μεταξύ του εξυπηρέτη και του πελάτη, και μεταξύ του εξυπηρέτη και του περιβάλλοντος οπτικοποίησης.

Το περιβάλλον οπτικοποίησης, που αποτελεί τη βάση του συστήματος, είναι βασισμένο σε συστατικά (modules). Αυτό σημαίνει ότι η λειτουργία του βασίζεται σε ένα σύνολο συστατικών, που είναι ανεξάρτητες μεταξύ τους οντότητες, και χρησιμεύουν στο να εκτελούνται οι λειτουργίες που μπορούν να εφαρμοστούν σε μια αναπαράσταση ενός τρισδιάστατου αντικειμένου γραφικών. Τέτοιες λειτουργίες είναι:

(1) να φορτώνεις ένα αντικείμενο από ένα αρχείο,

(2) να εφαρμόζεις μετατροπές πάνω στο αντικείμενο,

(3) να αποθηκεύεις το αντικείμενο σε ένα αρχείο, και

(4) να απεικονίζεις το αντικείμενο στην οθόνη.

Αυτά τα συστατικά έχουν σχεδιαστεί με βάση μια ενιαία μορφή, και κάθε συστατικό έχει ένα σύνολο από παραμέτρους, εισόδους και εξόδους. Οι παράμετροι καθορίζουν τη συμπεριφορά του κάθε συστατικού, και εξαρτώνται από τη λειτουργία για την οποία είναι υπεύθυνο αυτό το συστατικό. Οι είσοδοι του συστατικού είναι τα ορίσματα που είναι απαραίτητο να πάρει το συστατικό για να εκτελεστεί. Συνήθως πρόκειται για ένα αντικείμενο που προέκυψε από τις μετατροπές κάποιου άλλου συστατικού. Οι έξοδοι ενός συστατικού είναι τα αντικείμενα που προκύπτουν κατά την εκτέλεση άλλων συστατικών. Πολλά συστατικά συνδεδεμένα κατάλληλα μεταξύ τους συγκροτούν ένα σχήμα που μπορεί να εκτελέσει πολύπλοκους υπολογισμούς στα υπό επεξεργασία αντικείμενα. Το σχήμα αυτό μπορεί να εκτελεστεί, ή να αποθηκευτεί για να χρησιμοποιηθεί αργότερα για περαιτέρω επεξεργασία.

Ο εξυπηρέτης, που παίζει το ρόλο του διαμεσολαβητή ανάμεσα στο περιβάλλον οπτικοποίησης και τον πελάτη, είναι υπεύθυνος για την παροχή των απαραίτητων δεδομένων που στον πελάτη ώστε να λειτουργήσει και να προσφέρει τις

υπηρεσίες του στον απομακρυσμένο χρήστη. Τις απαραίτητες πληροφορίες ο εξυπηρέτης τις παίρνει από το περιβάλλον οπτικοποίησης.

Ο πελάτης είναι το κομμάτι του συστήματος με το οποίο έρχεται σε επαφή ο τελικός χρήστης. Αρχικά παίρνει από τον εξυπηρέτη τη λίστα με τα διαθέσιμα συστατικά στο περιβάλλον οπτικοποίησης. Στη συνέχεια ο χρήστης μπορεί να αναπτύξει ένα σχήμα βασισμένο στα συστατικά αυτά, αλλάζοντας τις παραμέτρους τους, και συνδέοντάς τα κατάλληλα. Αφού ολοκληρώσει το σχήμα του, μπορεί να το εκτελέσει και να πάρει τα αποτελέσματα. Αν στα αποτελέσματα υπάρχουν και δεδομένα προς απεικόνιση, τότε ο πελάτης τα φορτώνει και τα εμφανίζει στην οθόνη. Τα υπόλοιπα αποτελέσματα, όπως αντικείμενα προς αποθήκευση σε αρχείο, διατηρούνται στον υπολογιστή του εξυπηρέτη, αφού δεν είναι απαραίτητο να μεταδοθούν στον πελάτη. Αντί να εκτελέσει το σχήμα του, ο χρήστης μπορεί επίσης να το αποθηκεύσει για μελλοντική επεξεργασία. Το αποθηκευμένο σχήμα μένει στον υπολογιστή του εξυπηρέτη, έτσι ώστε να μπορεί να φορτωθεί κάποια άλλη φορά, από τον χρήστη που το αποθήκευσε, ή ακόμα καλύτερα, από άλλους χρήστες που θέλουν να το χρησιμοποιήσουν. Ο πελάτης καταφέρνει αυτές τις λειτουργίες, της αποθήκευσης και φόρτωσης ενός σχήματος, με κατάλληλη επικοινωνία με τον εξυπηρέτη. Το περιβάλλον οπτικοποίησης δεν συμμετέχει εδώ.

Το πρωτόκολλο για την επικοινωνία μεταξύ του εξυπηρέτη και του πελάτη, και μεταξύ του εξυπηρέτη και του περιβάλλοντος οπτικοποίησης, καθορίζει τη μορφή που έχουν τα δεδομένα που μεταδίδονται, το περιεχόμενο των δεδομένων, και τις περιπτώσεις που χρειάζεται μετάδοση δεδομένων. Η επικοινωνία οργανώνεται ως εξής:

- Όταν ο πελάτης συνδέεται, ζητά από τον εξυπηρέτη τη λίστα με τα διαθέσιμα συστατικά
- Ο εξυπηρέτης καλεί το περιβάλλον οπτικοποίησης για τη λίστα και τη στέλνει στον πελάτη
- Ο πελάτης καλεί τον εξυπηρέτη για να εκτελέσει ένα σχήμα
- Ο εξυπηρέτης καλεί το περιβάλλον για να εκτελέσει το σχήμα, και στέλνει τα αποτελέσματα στον πελάτη
- Αντί για εκτέλεση, ο πελάτης μπορεί να στείλει το σχήμα στον εξυπηρέτη για αποθήκευση

- Ο πελάτης μπορεί επίσης να φορτώσει ένα σχήμα. Ζητά τη λίστα με τα διαθέσιμα σχήματα από τον εξυπηρέτη, και τη λάβει ζητά ένα σχήμα. Ο εξυπηρέτης στέλνει αυτό το σχήμα.

Τα δεδομένα βρίσκονται πάντα σε XML μορφή, και περιέχουν μόνο την πληροφορία που είναι απαραίτητη για τη λειτουργία του όλου συστήματος. Κάποιες παράμετροι από το αρχικό περιβάλλον οπτικοποίησης έπρεπε να παραλειφθούν, και να αντικατασταθούν με σταθερές τιμές, για να επιτύχουμε ομοιομορφία στις παραμέτρους και να μειωθούν τα μεταδιδόμενα δεδομένα.

Η απόδοση του συστήματός μας είναι περισσότερο από ικανοποιητική. Ο χρόνος που χρειάζεται για να εκτελεστούν οι παραπάνω λειτουργίες ήταν ικανοποιητικός, και η κατάσταση του δικτύου και των εμπλεκομένων υπολογιστών δεν έδειξαν να επηρεάζουν τη γενική απόδοση. Το σύστημα απέδωσε καλώς ακόμα και σε περιπτώσεις με αυξημένη κίνηση στο δίκτυο, ή χαμηλή διαθεσιμότητα πόρων στους υπολογιστές του εξυπηρέτη και του πελάτη. Συμπερασματικά, σχεδιάσαμε και υλοποιήσαμε ένα σύστημα που παρέχει απομακρυσμένη πρόσβαση στο σύστημα οπτικοποίησης μέσω του Internet, με την ελάχιστη δυνατή επικοινωνία, και το οποίο δουλεύει καλά σε διάφορες καταστάσεις των εμπλεκόμενων πόρων.

Table of Contents

Preface 9

Chapter 1 Related Work 11

Chapter 2 Overview 19
2.1. Introduction 20
2.2. A Client-Server System for Internet Access to a Modular Visualization Environment 21
2.2.1. The Modular Visualization Environment 21
2.2.2. Remote Access to the Environment 23
2.2.3. Internet Client for Web Based Operation 24
2.2.4. File sharing 26

Chapter 3 The Modular Visualization Environment 28
3.1. About Computer Work Environments 29
3.2. Features of a Computer Work Environment 29
3.3. The Modular Visualization Environment 30
3.3.1. About the Modules 32
3.3.2. Creating a scheme 33
3.3.3. Saving and running a scheme 33
3.4. Examples of visualization 33
3.4.1. Examples of visualization of (valuated) points 33
3.4.2. Examples of the visualization of triangles 35
3.4.3. Visualization and operations over volumetric data 37
3.4.4. Visualization of slices 38
3.4.5. List of example data 39

Chapter 4 The Server 40
4.1. About the Server 41
4.2. Interaction between the MVE and the Server 41
4.3. Interaction between the Server and the client 45
4.4. MVE XML files description 47

Chapter 5 The Client 53
5.1. About the Client 54
5.2. JAVA Applets 54
5.3. The Client's GUI 55
5.4. Runtime of the client 56
5.5. Operations on the scheme 60
5.5.1. Saving a scheme 61
5.5.2. Loading a scheme 62
5.5.3. Executing a scheme 64

Chapter 6 Experiments – Performance 66
6.1. First Experiment 67
6.2. Second Experiment 77

7. Conclusions – Future Work 84

References 86

Appendix 1 Source Code of the Client 89

Appendix 2 Source Code of the Server 105

Preface

In the last few years the computational power of computers has increased considerably, while the cost of a powerful computer has decreased. People now can afford to have a personal computer at home or at their office, with such computational power that no one could afford some years ago, except from large organizations, governments and the military.

But still, as the computational power of computer has increased, the demand of the applications for speed and system resources has increased also. One of the tasks that require large amounts of system resources and computational power is that of 3D graphics processing and visualization.

A 3D graphics object may be represented as a set of points and a set of vertices that connect these points with each other in order to form a complex or simple surface in the three-dimensional space. Representations such as these may require large amounts of memory to be stored, and computations over these data may be very demanding in memory and execution time. Thus to process and visualize 3D graphics objects, we should consider using a powerful computer, with large amounts of memory available, and high computational speed.

Furthermore, if we design a system that provides services such as 3D graphics processing and visualization, to more than a single user, these demands are multiplied as the users of the system perform computations on it. So a very powerful computer is needed, where all the necessary computations can be performed.

One solution to the above mentioned problem would be to design a system that could execute distributed in a cluster of computers, so that the workload would be balanced among the involved computers. This would allow more computations to be performed concurrently, and the performance of the system could increase significantly. Also, more users can work at the same time, and even collaborate to perform more complex computations.

However, even if we have a cluster of powerful computers to do all the computationally intensive tasks, this system is usually available only to those that have physical access to these computers. If we want to provide a system to remote users as well, we should add support for remote access through the network. And the most suitable network to provide this support is the Internet, as it is the most

commonly used network available, and users around the world may access remote computers from any place at any time. So, what we propose as a solution for the problem, is to design a client – server system that provides remote Internet access to the visualization environment, whether the latter can runs on a cluster of workstations or a single machine.

One issue that normally comes up by providing network support for remote access to the environment, is that of the network workload between the clients and the server, which is a consequence of the fact that the objects we process require large amounts of data to be stored. This means that if a remote user wants to process a 3D graphics object, the user has to transmit the data over the network, perform the computations on the object, and send the modified object back. If the remote user has a slow network connection or a computer with limited computational capabilities, he would not be able to perform these tasks.

This is why we propose to design a client – server system that provides for remote access to users through the Internet, where the computations are performed strictly in the server computers, and not on the clients. We also want the network communication to be as limited as possible, so that the system does not depend on network status, and even clients with slow network connections can perform their computations easily.

So the solution we propose is to design a client – server system that uses a reduced and efficient protocol for the communication between the client and the server, and where the computations are performed at the servers' powerful computers. The only data the client has to acquire from the server is a list of the available transformations and modifications on the objects, and not the objects themselves. The remote user may then create a scheme that will give instructions to the server in order to perform the complex computations on behalf of the user.

Chapter 1

Related Work

The design of a Collaborative Internet Client for Remote Access to a Visualization or Virtual Environment involves several interesting research issues. It includes the details on the collaboration between the environment's users, and the actions a user can take during a session. Some of these issues have been studied by several projects, which are presented in this section.

In this work we will describe the design and the implementation details of the client – server system mentioned above, as well as the details of the protocol that is to be used for the communication between the client and the server.
In the rest of this chapter we present a survey of related projects and studies that deal with the same or similar problems such as ours. In chapter 2 – the overview of our study – we describe the details of the issues that have to be taken under consideration during the design and the implementation of the system. In the third chapter we provide a brief description of the modular visualization environment, which is the environment we used to perform the computations on the 3D graphics objects. In chapter 4 we describe the details of the server, and a protocol to be used for the communication between the server and the client, and also between the server and the environment. In chapter 5 we describe the details and the features of the client. In chapter 6, we measure the performance of our implementation through experiments we performed and compare with other system that provide similar services as we do. In chapter 7, we make our conclusions and propose how this work could be extended, and at the end we provide an appendix with the source code of the client and the server. Some projects that study the design and implementation of collaborative environments are presented bellow.

In the University of Pittsburgh, a team of scientists has designed a collaborative environment they called Networked Intelligent Collaborative Environment, or NetICE [16]. It is a typical virtual reality environment, for general use and research purposes, and it allows people to communicate from anywhere at any time, with the use of an intelligent network and smart terminals. The intelligent network and the smart terminals together create a virtual link between two users so that they think they are communicating face-to-face. The intelligent network is the bridge that connects all the smart terminals together and coordinates their communications. The smart terminals are responsible for delivering an immersive environment to the end users by employing various speech and image processing techniques and 3D computer graphics animations.
The intelligent Network contains a centralized multipoint control unit (MCU) to which each smart terminal sends data streams, which can be either the avatar’s positions and body movements, or audio packets. The network combines the data sent

from all terminals to visualize the environment and playback the audio data produced by the users.

The user can navigate freely in the NetICE conference room and see other avatars' navigation as well. In addition, a client's smart terminal is able to detect collision with other clients in the virtual room so that the client is prohibited from getting too close to other clients while navigating.

The environment is still under development, and new features are to be added, such as the support of clothing for the avatars, and the ability to set the facial features according to an image scan of the user's face, in order to make the users easier to recognize.

Another collaborative environment was designed by a research team in the MIT, and it supports distributed CAD over the Internet [3]. According to the designers, the next generation of engineering support tools such as CAD/CAE systems will include integrated and distributed CAD/CAE systems with product and process models, coupled with design rationale information that is shared, reused and merged with other design processes as part of the collaborative design endeavor.

Their system presents a design rationale based framework to assist geographically distributed designers to collaborate remotely. The proposed framework allows for computer support to multiple designers in the area of active design rationale capture for reusability, by using:

1. A computer-supported design rationale model to capture design rationale. The model they propose to utilize is called Design Recommendation and Intent Model Extended for Reusability (DRIMER)
2. Collaborative tools for handling team interactions over the Internet, and
3. Case-based reasoning mechanisms for organizing and analyzing design artifacts and rationale.

In order to improve the ability to represent, capture, and reuse design rationale in collaborative settings, the following focus areas have been identified:

1. Creation of design rationale model and tools
2. Integration of the design rationale model with case-based reasoning principles
3. Providing a collaborative framework for the overall architecture

A prototype implementation based on this framework is still under development.

In another study at the University of Ioannina, the researchers have designed an open client-server system for remote interactive editing of 3D scenes of geometric objects [2]. The system is built upon a high level extensible protocol for communicating geometric information between a downloadable light-weight client that runs on any commercial platform and a server that usually runs on a powerful dedicated engine. The Extensible Geometry Transfer Protocol (EGTP) described aims at minimizing the information communicated between the light-weight client and the server, and allowing for collaborative concurrent editing. The server resident software is application specific and may include a solid modeling system, a geometric constraint solver, ray tracing software and various auxiliary components. Typical application of this client-server architecture include collaborative computer aided design, photorealistic entertainment graphics, and virtual reality scene development.

A different type of collaborative environment is that of the OpenDesign project [4]. It is an open user-configurable project environment that supports distributed collaborative design and execution on the Internet. The environment is created by configuring a generic client for a specific project. This is in contrast to an implementation of a project-specific client-server architecture.
The major goal of OpenDesign is to allow users to configure their environment:

- by choosing the best or the most affordable tools for each design tasks – without having to install them on local host
- by choosing the most effective sequences of tasks to be executed, not only for manual one-task-at-a-time execution but also for scheduling any number of tasks for automated execution
- by choosing the hierarchy of data structures and revision control most appropriate for the project-at-hand – and having it readily accessible on the Web
- by creating and storing preferences on the modes of collaboration among the members of a project team.

The approach to devising the OpenDesign environment reflects its goal as listed above. Rather than implementing a dedicated client/server environment where software developers make most of the choices of how the environment is to be used by designers, we rely on a generic client interfaced to a generic server and let the designers/users implement the environment themselves by writing a simple configuration file and a set of encapsulation scripts. As long as each point tool also can be invoked remotely with a command-line script rather than the nominal GUI, demos show that a number a number of project-specific environments can be readily created by re-configuration of the generic client.

The researchers in the NCSU have worked on a class software project that was part of a recent experimental graduate course on Frontiers of Collaborative Computing on the Internet [8]. They chose Tcl/Tk to facilitate rapid prototyping, testing, and demonstrating all phases of the project. The major milestones achieved during this course are:

- rapid proficiency in Tcl/Tk that allowed each student to manipulate data and widgets, apply socket programming principles, and create a progression of client/server applications, from textbook cases to a unique client/server architecture prototype – driven by and matched to a well-defined collaborative project driver.
- universal server that supports any number of user-configurable clients, each accessible through a Web-browser on a Mac, Windows, or UNIX platforms. Prototype client configurations include:
    - collaborative document composition
    - collaborative Tcl/Tk debugging and compilation, and
    - collaborative design workflow

According to the researchers in the NCSU [9], traditionally, a stand-alone application is rendered collaborative for members of a team either by sharing its view or by re-writing it as a collaborative client. However, it may not be possible to anticipate in advance all preferences for collaboration, hence such a client may appear confusing to some of the team members.

They propose a novel client/server architecture for tk-based applications: rendering any stand alone client collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The CollabWiseTk toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration. It redefines the existing tk commands such that the entire tk widget set is rendered collaborative for use with multiple users.

Some other collaborative environments have been designed with the use of Virtual Reality. Collaborative Virtual Environments (CVEs) [18] are the VR application to product and system design activities, allowing the viewing and review of entire systems assemblies and parts. The technology of Collaborative Virtual Environments aims to transform today's computer networks into navigable and populated 3D spaces that support collaborative work and social play. CVEs are virtual worlds shared by participants across a computer network. Participants are provided with graphical embodiments called avatars that convey their identity, presence, location, and activities to others. They are able to use these avatars to interact with the contents of the world and to communicate with one another using different media including audio, video, graphical gestures, and text.

CVEs can be seen as the result of a convergence of research interests within the VR and computer-supported cooperative work (CSCW) communities. Within the CSCW community CVEs represent a technology that may support some aspects of social interaction not readily accommodated by technologies such as audio and videoconferencing and shared desktop applications. CVEs may enable participants to discuss and manipulate shared 3D models and visualizations in such a way that each can adopt their own viewpoint and can naturally indicate to others where they look and point.

General Motors, for example, provided fertile ground for implementing collaborative technologies for designing cars [6]. With the use of a Collaborative Virtual Environment, in a dark room at the North American Design center of General Motors (GM), a discussion is going on around a vehicle concept. Opposite the reviewers, a detailed, full-size car spins on a turntable on a sun-lit patio. At the press of a button, the patio changes to a winter scene with the bare trees reflected in the car's surface.

Next, the car stands alongside the current vehicle brand line, or semi-transparently overlays its predecessor to show the design evolution.
Operations that were formerly impractical or expensive (such as reviewing vehicle mock-ups outdoors) are becoming commonplace in the automotive industry through the use of virtual environments. In production use at GM since 1995, these venues have become collaborative meeting rooms, and are now linking global working groups together electronically via a shared virtual model.
Networked virtual environments share data and control. GM has multiple remote sites connected by the company intranet. Four of these contain a CAVE [13] and a Wall in the same room so that virtual vehicle interiors and exteriors can be viewed together if desired. The principal data structure in these display systems is the scene graph, which holds the parameters and relationships among the objects to be rendered graphically. With a programming interface allowing dynamic modification of the data over a network, the scene graph can be shared in several ways: Among applications and the display system; among two or more display systems in the same room; and among geographically distributed display systems.

Another Virtual Reality Environment was designed in the University of Illinois Chicago, where the researchers elaborated the traditional VR application to architectural design. Their ideas of providing multiple perspectives for collaborative design are applied to CALVIN (Collaborative Architectural Layout Via Immersive Navigation), a prototype system they implemented [17]. CALVIN implements multiple camera perspectives and allows multiple participants to collaboratively design in a shared architectural space.
One of the obvious affordances of VR is its ability to depict environments from an ‘inside-out’ perspective. That is, the viewers are placed in a position where they are physically immersed in the environment. This has been leveraged by many researchers to produce three-dimensional walk-throughs of architectural spaces. These implementations have been very successful because they offer clients the ability to tour a building design before it is actually built.
This is useful in finding design problems before a great deal of money is spent in actually building the structure. However, in order to visualize the end object in the Virtual Environment, the CAD model must first be constructed. This means that VR

is not yet being used to support creative task of design and problem solving, since it can only render the final result of the design process.

One of the projects aims is to provide an environment that introduces VR early in the design process. This can be achieved by using a standard set of objects that can be added or removed from the environment at any time. Another feature that would help is a 3D sketching interface that could allows designers to quickly turn their hand-drawn sketches into rough 3D studies. These studies can then become additional objects in the environment, allowing the redesign and problem solving of the object under development.

Another aim for the designers of CALVIN is to be able to build persistent Virtual Environments, so that the users may leave the space and re-enter to continue their work and collaborate with other users. This would give the designers the opportunity to enter the world any time they have new inspirations for possible design solutions, and without following any schedule that limits their efficiency.

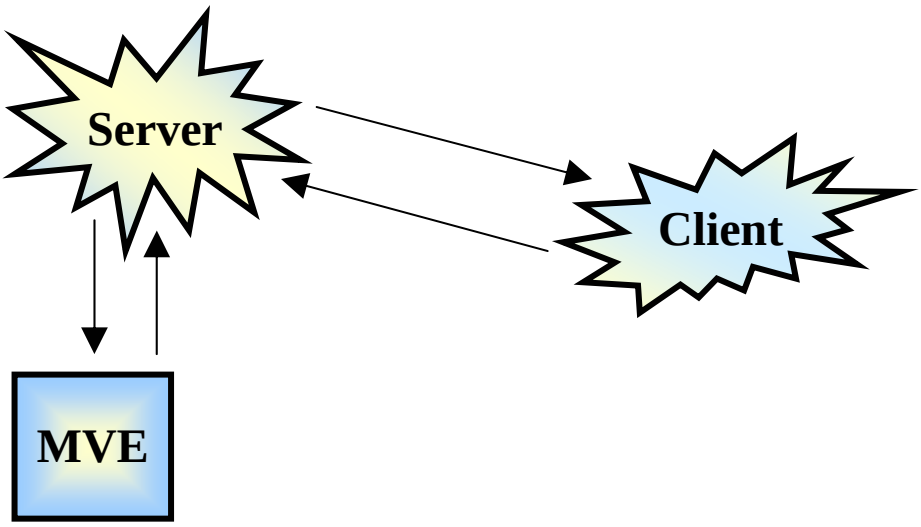
Another application of CVEs is that of simulators used for the training of people under dangerous circumstances, such as pilots, astronauts, divers, etc [14]. Apart from the training simulators, CVEs are used in 3D multiplayer games, where several players enter a virtual world and play with each other, sometimes even if they are dispersed all over the world. Experiments being conducted on molecules can easily be visualized using a CVE [15]. Many users of this environment are able to share the vision of the virtual world, and to collaborate on the tasks of understanding a complex molecular structure.

But a Collaborative Environment does not always have to include a Virtual Reality Environment. Many applications provide networked collaboration support. By using this feature, users from anywhere can communicate with each other at any time to develop a scheme of any type, sometimes a CAD design, or an image, or an 3D graphics object, or whatever they might think of.

Chapter 2

Overview

In this chapter we will provide general information about our project and the issues we encountered during the design and implementation, such as the architecture of the system, and the specifications of the system parts.



2.1. Introduction

The aim of this work is to create a visualization environment that has three basic features: it is Modular, it provides Remote Access, and it provides file sharing. A Modular Visualization Environment is a working environment, where a user can develop a scheme that consists of a set of modules that are connected to each other, in order to complete his task. The term ‘Remote Access’ suggests that this environment should be able to act as a server, which accepts orders from clients, who want to create and compile their schemes. The environment itself can be distributed or localized – that is not of our main concerns – and it should have access to the network, because the clients may be located anywhere around the world. The term ‘Collaborative’ suggests that the users of this environment can have a direct or indirect communication with each other, so that they can work together as a team, in order to develop or to share their schemes.

2.2. A Client-Server System for Internet Access to a Modular Visualization Environment

2.2.1. The Modular Visualization Environment

A Modular Visualization Environment is an environment that is based on a set of modules, in order to create, compile and execute schemes. Each Module has a set of parameters, inputs and outputs, and it is independent from other modules. These modules are responsible for several tasks, such as loading object representations, applying computations, filtering, altering, and saving them, and the most essentially – rendering them. Several modules connected to each other properly form a scheme, which can be saved or executed.

A Module consists of a set of subroutines, procedures or functions of a program, which may be compiled and executed on its own, and not as part of the main program. Each module is responsible for a specific task and it does not depend on the rest of the modules that are present on the current scheme. This means that a module can be executed individually and, even if some other modules fail to run, the module in question does not fail, as long as it does not expect any input or lie in the same path with the module that has failed to complete its task. The modules that comprise the scheme are connected to each other in a way that the output of one module becomes the input to the next, and so on. Some of the modules have to wait for the proper input from other modules, so they have to be executed serially. Some other modules though, depending on the scheme, do not depend on others, so they can be executed in parallel. This way, the environment supports parallel execution. Furthermore, some modules may be executed on one machine, whereas some others on another, and the outputs of the modules on the first machine can be propagated as inputs to the modules on the next machine. Although there are several ways for this to be accomplished, it is not of our main concerns.

In the figure 2.1 we can see an example of a module. This example shows a module whose input is an object, in this case a set of frequencies. This module applies a filter of some kind to the object and, as output, sends the filtered object as input to the next module for further processing, and so on.

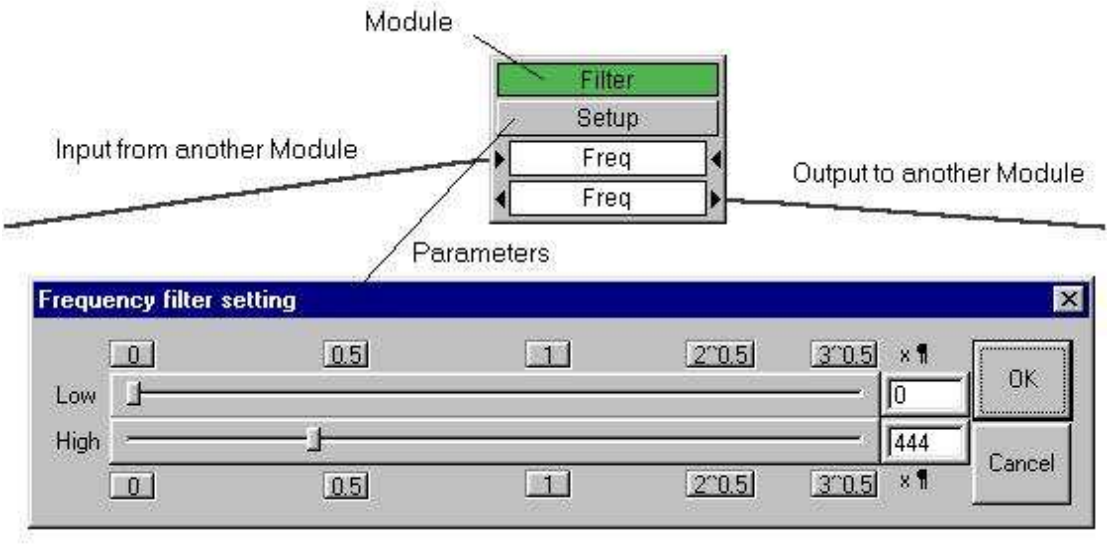


Figure 2.1: An example of a module, filtering a set of Frequencies.

The parameters of the module depend on the module itself. For example, in the case of the filtering module in figure 2.1, the parameter is the range of the frequencies to filter. If the module were an image loader, its parameter would be the filename for the image to be loaded. The number of inputs and outputs may vary from module to module as well. A module depends on its inputs and its parameters only. If the inputs of the module are not connected with corresponding outputs of other modules, the module will not be able to run successfully. Therefore we have to establish the connections between the modules in order for the scheme to be complete. Obviously the output of one module must match the input of the next, so that the modules can operate correctly. In the example above, the filtering module expects a set of frequencies as input and provides another set of frequencies as output. If the input it receives is of a different type, or the output becomes input as a different type, there will be an inconsistency that may cause the entire scheme to crash.

As we can see, the modules have formed a complete scheme in order to be executed successfully, and the parameters and connections of the modules determine the outcome of the whole scheme.

2.2.2. Remote Access to the Environment

Up to now, the description of the Modular Visualization Environment suggested that the user has access to the local machine where the environment is installed. What we want is to provide remote access to the environment, so that the users may use it through a network connection. To achieve this, we had to make some modifications to the initial environment in order for it to accept network connections. The Modular Visualization Environment itself was basically a scheme editor, where a user could add modules to the workspace of the editor by dragging them from a list with the mouse. The user could save the scheme or run it by pressing corresponding buttons, but generally he had to have physical access to the machine in order to operate the environment.

The first thing we had to do was to enable the environment to accept command line arguments, so that it could be called from an external application. Our associates at the University of West Bohemia, who had implemented the initial Modular Visualization Environment, undertook this part of the project, and converted the MVE to accept command line arguments. The resulting application was MVE_Remote that could load and execute schemes as command line arguments.

But still the environment did not support a socket interface so that it could communicate with other applications via a network. That was the reason why we had to develop a middleware for the MVE, a server application that could act as an intermediate between the MVE_Remote and the clients who wished to connect to it.

The role of this server is to accept connections from clients, receive their schemes, save them to its local hard disk and to call MVE_Remote in order to execute them. MVE Remote is unaware of the clients that are connected and it only provides its service when it is called from the server. The server accepts the connections from the clients, informs them of available modules on MVE (MVE_Remote is called once at this point), accepts schemes to be saved, accepts schemes to be executed (MVE_Remote is called for this), and sends previously saved schemes to the clients after a load request. The client, of course, does not know whether the server or the MVE is handling his requests. The only demands the client has is to receive a list of available modules in order to be able to save his scheme on the server's hard disk, to load schemes from the server's disk, and to execute them.

2.2.3. Internet Client for Web Based Operation

The next feature we want our client to have is that of a web-based application, meaning an Internet client. The aim of this project is to provide remote access to the Modular Visualization Environment in a manner that suits any user, regardless of the platform he is using or the resources that are available on his machine. Furthermore, the client should run within a web browser, so that a user can load his client through the Internet. For these two reasons, we designed the client to be a Java Applet. Java classes can run on any platform that supports Java, without needing to be recompiled for the new platform. Most known platforms today support Java, and the Java Runtime Environment is available for distribution on the Internet for free. Another asset of Java Applets is that the classes do not have to be installed on the local machine where the user wants to run his client. The classes are downloaded instantly for the user and are executed by the web browser, without the necessity of making changes to the system.

However, there are some problems that are caused from the fact that the client is a Java applet. First of all the client has no write access to the disk where it is being executed. There are some ways to overcome this issue, but they mostly require modifications on the local machine of the client, like applet access policies etc. We want the user to be able to execute the client without having to modify the settings on his machine. What we chose to do is to keep all the buffers in memory, and renounce the privilege of having access to the client's storage medium. The amount of data we keep in memory is not that much anyway, so there will not be any problem with eating up all the memory resources.

Another issue that could cause a problem for the client is the communication with the server. The client uses a socket to connect to the server. The problem is that the applet is allowed to open a socket only to the machine from which it was downloaded. This means that if the server and the MVE are located on a different machine than the one having the client's classes, the socket will not be accepted for security reasons, and the connection will not be established. To overcome this problem, we had to install the client's classes, along with the web server, on the same machine where the MVE and the server are installed. This fact increases the workload on the server's machine, since it has to accept http connections from users who want to download and execute

the client. However, it is the simplest way and the workload on the server's machine is still acceptable.

There are still some issues that arise from the fact that the client is a Java Applet, but we cannot provide for them, as the necessary measures have to be taken by the user of the client at his local machine. The first thing is that most distributions of web browsers do not include the JRE in the main package, therefore if a user wants to run the client and its machine does not support Java or the Java plugin is not installed in the web browser he uses, the client will not execute. In order to run the client, the least the remote user has to do is install the Java Runtime Environment and the Java plugin for web browsers. These packages can be downloaded from the homepage of SUN Microsystems (java.sun.com), which is the developer of Java. We used JRE 1.3 and JRE 1.4 to compile the classes of the client.

Our associates at the University of West Bohemia are still working on the conversion of the MVE, so the existing modules that export data for rendering do not support VRML format. When the conversion of the modules is complete, MVE_Remote will be able to export VRML objects, so the web browser where the client runs has to be able to load them. For this, the user has to enhance his web browser with a VRML plugin. Several VRML plugins are also available on the Internet. The only output results we got during this study was static images, which did not need any special plugin in order for them to be loaded on the web browser.

A summary of the issues stated above is shown on the Table 1.1:

Assets	**Problems**	**Solutions**
Can run on any platform	Does not have access to local resources of the machine it is executed	Keep everything in memory. The memory needed is not that much.
No need for installation	Plugins need to be installed for the web browser	The user has to install the JRE.
Available to any Internet user	Can only connect to the machine it was downloaded from	Run the server on the same machine.

Table 2.1: Advantages and disadvantages of the client being a Java applet.

2.2.4. File sharing

Another feature we want our client to have is that of the ability to share files. This means that the files a user has created could be used from other users as well. In order for a user to share his scheme with other users, the only thing he has to do is to make it available to the public, so that others may retrieve it and process it further. Our server supports that kind of file sharing, since all users save their schemes in the same directory (on the server's machine). This way, any one can freely load another user's scheme at any time, as if it were his. A matter of sharing policy comes up at this point, since some users would have a problem with other users loading their schemes. To deal with this issue, we could organize the files according to the users, or even groups, so that a user could choose his partners from a list of users, and allow some of them to have access to his files. This means that each user should identify and authenticate himself by using a username and password in order to login to the system.

The client and the server could support user authentication, and attributes could be set on the schemes from the server, so that each scheme could be readable or even writable from users other than the user who created it. Then, the user who created the scheme could set the corresponding flags concerning his scheme, and give read or write access to specific users, or users in the same group as him, or even to all users. Therefore, when another user wants to have access to a particular scheme, he could enter his username and password, and if he has the right to access this scheme, he is free to load it.

This way we can achieve a fully organized sharing system, with limited permissions to files, users or groups. It is also flexible, since the user can choose whom he grants access to and to whom he denies it.

The implementation of this system does not require major changes to the server's source code, nor to the client's. The only requirement is a login process and an elementary database to keep the usernames, passwords, and privileges of users to other users' files. The attributes of the files could be set in this database as well.

So far, the client and the server do not have to communicate continuously, since the whole session is based on a small amount of data, so when this amount of data is exchanged, the communication is unnecessary, and the connection is idle. If two users

would like to cooperate in developing a scheme, then the two clients should be in constant connection, and the one user should be notified for each and every move the other user makes. This is similar to the Virtual Environments described in [5,6], and the design of the whole project should be reevaluated, as the new feature would require a totally different implementation. In the case of the cooperation of two users, things might be a little bit simpler, since each move of the first user would have to be transmitted to the second, and vice versa. In that case, we could even use peer-to-peer communication between the users, so that they could communicate more effectively, and so that the workload of the centralized server is reduced. But even then we have to prioritize some actions made by the users over other actions.

For example, lets assume that two users have loaded a 3D object – for instance, a 3D cow model – and they want to process it together. The first user rotates the cow 90 degrees on the x-axis, keeping the head still (and moving the rest of the body). The second user rotates the cow 90 degrees on the z-axis, keeping the tail still. These two actions will have a totally different result, if applied in a different order. The cow will be in a different position in the 3-dimensional space. If we begin to apply filters and transformations on the object, things will become more complex. Some projects that dealt with this problem, chose to use case-based reasoning principles in order to reduce the variety of the results [3].

The previous, simple example was given for two users collaborating. If we try to generalize the scheme for more users, we will find it very difficult even to conceive.

Another way of implementing this feature is to let each user process his scheme individually and then to combine the schemes of the users according to some rules. For example, if a user applies a filter to an object and another user applies a different filter or a transformation to the object, we could merge these two modifications and return the final result. This, however, makes things even more complex, because it is difficult to monitor each user's actions individually, and the same problem as before still exists, meaning the variety of the results. In any case, Real-time collaboration between the users is an issue that could be researched on its own, since the problems one would encounter in the process would be difficult to overcome.

Chapter 3

The Modular Visualization Environment

In this chapter we will provide detailed information about the Environment we have used in order to design and implement our client-server application. The Modular Visualization Environment was designed and implemented by our associates at the University of West Bohemia of the Czech Republic Martin Franc and Michal Rousal as part of their PhD degree.

3.1. About Computer Work Environments

When people talk about a work environment on a computer, they might think of any environment that could provide the means for the user to deploy a scheme to be saved and/or executed. The term ‘Scheme’ addresses any form of implementation on a Computer platform, which means a computer program, an image file, a blueprint design, a file, and generally an object of any type. In our case, a scheme is a set of objects, the modules, which are connected to each other and form a complex set of computations to be applied on 3D graphics representations.

The simplest example of a work environment one could think of is a text editor, where we can write the source code for a program. The text editor represents the work environment, while the source code represents the scheme, which is to be saved and compiled.
An example of a working environment would also be a program for image, sound, or video processing, where we are provided with an object (an image, a sound, or a video stream), which we process by applying filters or routines to it, by altering its features etc.
Another example of work environments would be a CAD program, where we could develop a Computer Aided Design, such as a blueprint of any kind that could be used for further processing.

3.2. Features of a Computer Work Environment

A computer work environment should offer a set of features that conduce to the easiest interoperation for the user. In other words, the environment should provide the user with several functionalities, so that he may utilize it in the best way, in order to develop his scheme.
If we consider the example of the text editor, the features it offers to the user could be:

- The ability to compile the source code
- Searching and replacing parts of text
- Syntax highlighting of the source code

Some known text editors, such as Nedit in Unix or WinEdit for Windows, even provide syntax highlighting for more than one programming language, and can detect the current programming language used by examining parts of the source code being written.
Furthermore, text editors included in programming languages' distributions, like Forte for Java, provide field auto-fill, a feature that prevents the user from making mistakes and increases the typing speed, by filling in the name of the routine from a set of available alternatives.
Every kind of working environment on a computer has a corresponding set of features, which are designed to provide the user with a better look-and-feel.

3.3. The Modular Visualization Environment

The environment we used in order to develop and run our schemes is the MVE (Modular Visualization Environment). The name is self-explanatory for the operation of the environment. The user can manipulate an object using a scheme that is comprised of modules. In order for the user to deploy his scheme, he loads a number of modules into the workspace area, set the parameters of these modules, and connects them to each other, so that the output of one module becomes input to one or more other modules.

The environment of the MVE consists of a scheme editor, which is a workspace where modules could be loaded and processed. The main window of the MVE editor is shown at figure 3.1.

Figure 3.1: The Scheme Editor of the MVE.

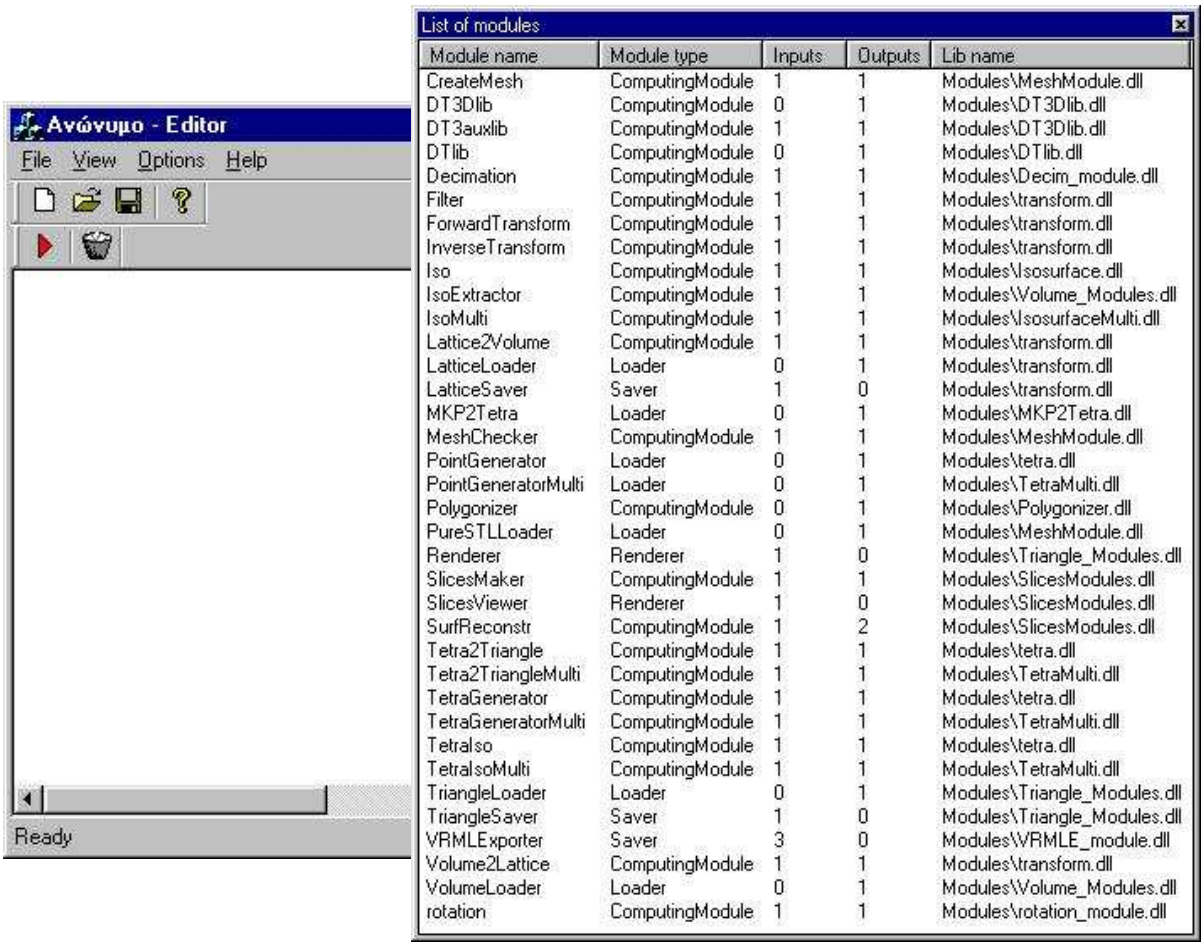
The first thing a user could do to develop a scheme, is to load a list of the available modules.

The modules are individual components, mainly procedures and subroutines, written by the implementor of MVE or another user. They come in the form of precompiled objects in *.dll's and are easily installed to the MVE. The programming language in which they are written is of no importance, since the modules are compiled individually and not as a part of MVE. Once a module is loaded to the system, it becomes available to all users, making the environment more flexible and operational.

3.3.1. About the Modules

There are several types of modules, depending on the task that they are responsible for. There are:

- Loader modules, which are responsible for loading an object from a file to the scheme
- Saver modules, which save the resulting objects to a file
- Computing modules, which apply filters to a loaded object, or process the object, and generally make changes on it
- Renderer modules, which undertake the task of the visualization of the object



Module name	Module type	Inputs	Outputs	Lib name
CreateMesh	ComputingModule	1	1	Modules\MeshModule.dll
DT3Dlib	ComputingModule	0	1	Modules\DT3Dlib.dll
DT3auxlib	ComputingModule	1	1	Modules\DT3Dlib.dll
DTlib	ComputingModule	0	1	Modules\DTlib.dll
Decimation	ComputingModule	1	1	Modules\Decim_module.dll
Filter	ComputingModule	1	1	Modules\transform.dll
ForwardTransform	ComputingModule	1	1	Modules\transform.dll
InverseTransform	ComputingModule	1	1	Modules\transform.dll
Iso	ComputingModule	1	1	Modules\Isosurface.dll
IsoExtractor	ComputingModule	1	1	Modules\Volume_Modules.dll
IsoMulti	ComputingModule	1	1	Modules\IsosurfaceMulti.dll
Lattice2Volume	ComputingModule	1	1	Modules\transform.dll
LatticeLoader	Loader	0	1	Modules\transform.dll
LatticeSaver	Saver	1	0	Modules\transform.dll
MKP2Tetra	Loader	0	1	Modules\MKP2Tetra.dll
MeshChecker	ComputingModule	1	1	Modules\MeshModule.dll
PointGenerator	Loader	0	1	Modules\tetra.dll
PointGeneratorMulti	Loader	0	1	Modules\TetraMulti.dll
Polygonizer	ComputingModule	0	1	Modules\Polygonizer.dll
PureSTLLoader	Loader	0	1	Modules\MeshModule.dll
Renderer	Renderer	1	0	Modules\Triangle_Modules.dll
SlicesMaker	ComputingModule	1	1	Modules\SlicesModules.dll
SlicesViewer	Renderer	1	0	Modules\SlicesModules.dll
SurfReconstr	ComputingModule	1	2	Modules\SlicesModules.dll
Tetra2Triangle	ComputingModule	1	1	Modules\tetra.dll
Tetra2TriangleMulti	ComputingModule	1	1	Modules\TetraMulti.dll
TetraGenerator	ComputingModule	1	1	Modules\tetra.dll
TetraGeneratorMulti	ComputingModule	1	1	Modules\TetraMulti.dll
TetraIso	ComputingModule	1	1	Modules\tetra.dll
TetraIsoMulti	ComputingModule	1	1	Modules\TetraMulti.dll
TriangleLoader	Loader	0	1	Modules\Triangle_Modules.dll
TriangleSaver	Saver	1	0	Modules\Triangle_Modules.dll
VRMLExporter	Saver	3	0	Modules\VRMLE_module.dll
Volume2Lattice	ComputingModule	1	1	Modules\transform.dll
VolumeLoader	Loader	0	1	Modules\Volume_Modules.dll
rotation	ComputingModule	1	1	Modules\rotation_module.dll

Figure 3.2: The list of currently available modules.

A typical list of the available modules is shown at the figure 3.2. As we can see, each module has a varying number of inputs and outputs. Apart from its inputs and outputs, a module has a set of parameters, which define the behavior of the module. For example, the 'LatticeLoader' module has zero inputs and one output. The parameters of this module comprise the name of the file that contains the object to be loaded, in lattice representation. The output of the module is the lattice representation of the object, which will become input to other modules for further processing.

3.3.2. Creating a scheme

To add the module to the scheme, the user just drags the module name from the list, inside the workspace area. A small window appears, representing the module, and the inputs and outputs are shown, along with the setup button, by which we can set the parameters of the module.

When the desired modules have been added to the scheme, we have to connect them to each other, so that the output of the first one becomes input to the next, etc. We can connect two modules by dragging with the mouse from a specified output to an input with matching datatype. The connection is established and a line appears, connecting the corresponding input an output.

This way we have developed a scheme, which loads a lattice representation from a file, applies a transformation to it – like Fast Fourier Transform – converting it into a set of frequencies, filters these frequencies, transforms them back to a lattice, and saves the output back to a file. This means we have filtered the object using several modules connected with each other, as shown in figure 3.3.

In the example of the figure 3.3 we can also see the setup dialog box of a module, where we set its parameters. Each module has its own set of parameters, and some modules – like the Inverse Transform in our example – may have no parameters to set.

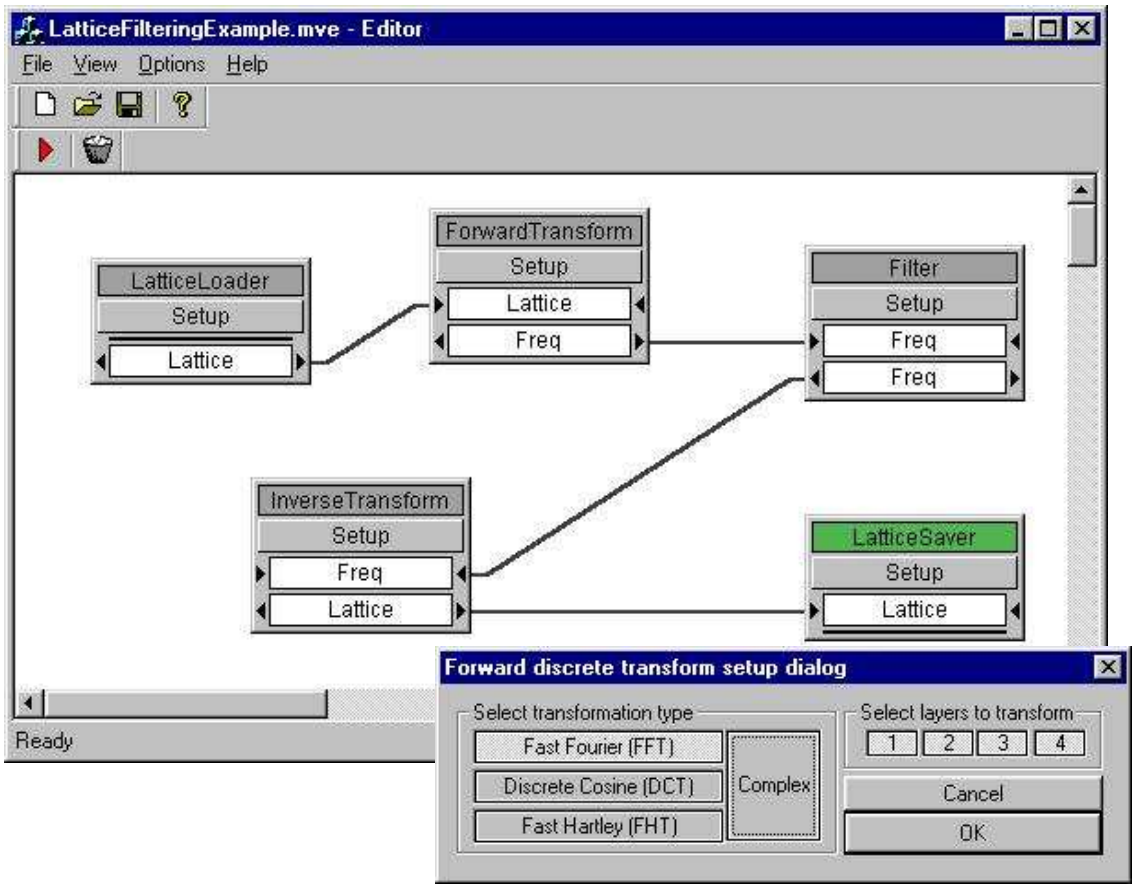


Figure 3.3: An example of a scheme. Applying a filter on a transformed object.

Needless to say, the types of input differ from module to module, and they have to match, for a connection to be established. The types of input and output available are the following:

- Points
- Triangles
- Tetrahedras
- Lattice
- Volume
- Freq
- FreqHart
- FreqCos
- XYZSlices
- Tetrahedra2
- Points2
- Slices

3.3.3. Saving and running a scheme

When the scheme is completed we can save it, or run it. The execution of a scheme can either be done locally, or distributed, since parts of the available modules may be located on different machines other than the local host. This provides a parallel execution of the scheme. However, our main consideration is the execution of the scheme, and thus we are not concerned if it is done locally or distributed.

3.4. Examples of visualization

3.4.1. Examples of visualization of (valuated) points

Points can be visualized either using tetrahedrons or triangles. The visualization depends on the modules used. Here are some examples.

In figure 3.4 we see an example of using Renderer for visualization of tetrahedra created from a set of points that have not been evaluated.

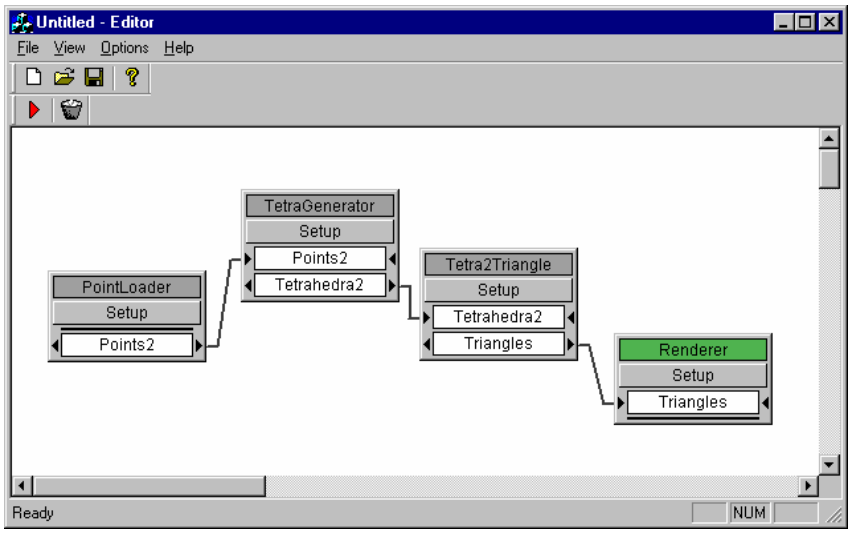


Figure 3.4: Visualization of non-evaluated points (tetrahedra)

An example of visualization of evaluated points using Renderer is showed in figure 3.5. Points are loaded and converted to tetrahedra using Finite elements method and then converted to the iso surfaces (triangle mesh) for rendering.

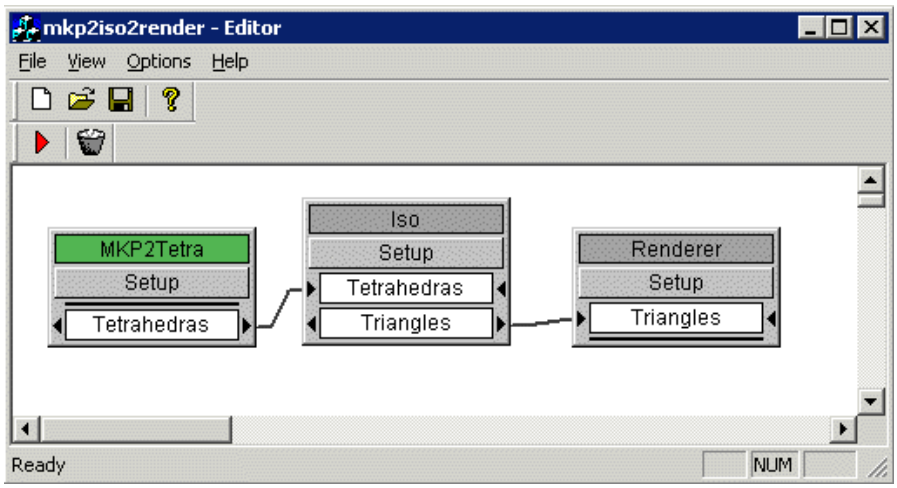


Figure 3.5: Example of using the MKP2Tetra module

In figure 3.6 and figure 3.7 one can be seen how the Delaunay triangularization (tetrahedronization) can be used in MVE. In figure 3.6 there is a 2D case : triangularization is done de facto only in two dimensions, but each vertex can have height (value). In figure 3.7 there is a 3D case.

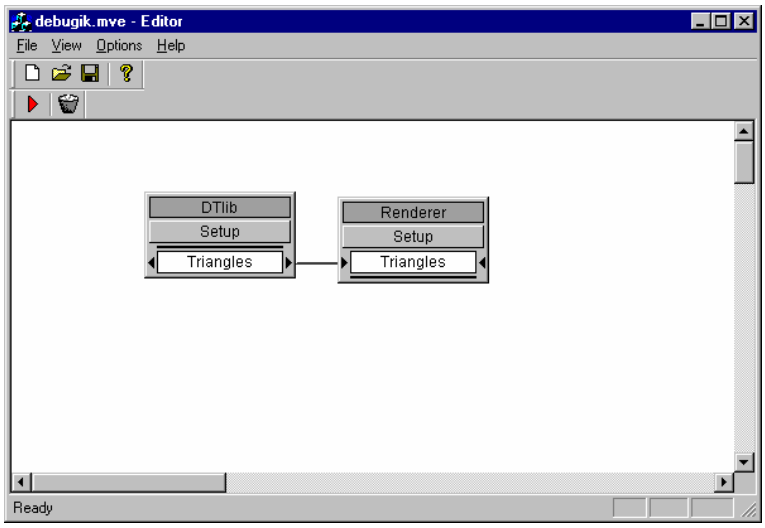


Figure 3.6: 2D Delaunay triangularization

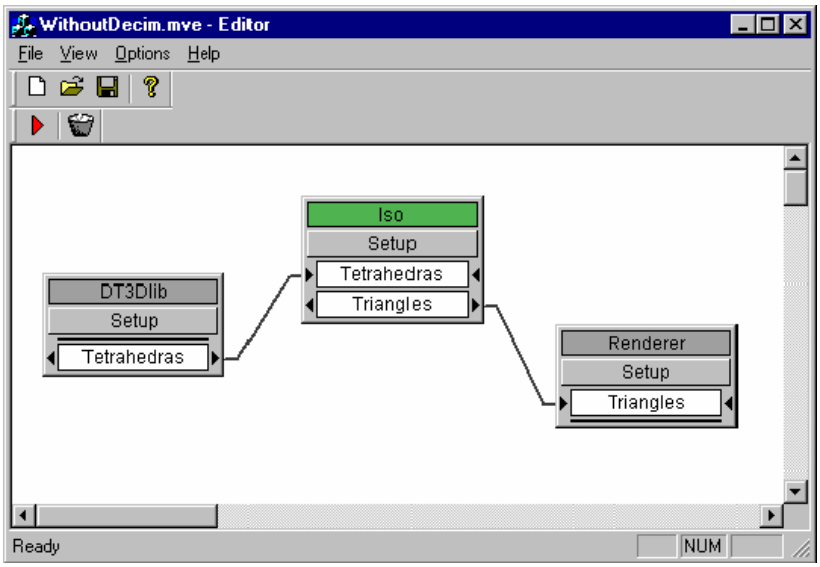


Figure 3.7: 3D Delaunay tetrahedronization

3.4.2. Examples of the visualization of triangles

Visualization through triangles is the most common used method. Triangles can be directly visualized using `Renderer`.

The common way for visualization of triangles is displayed in figure 3.8. Note the decimation module, which is used to reduce the number of triangles.

In figure 3.9 one can see a scheme for adding some information about mesh to the triangles such as adjacency, normal vectors etc. This information can be used by other modules.

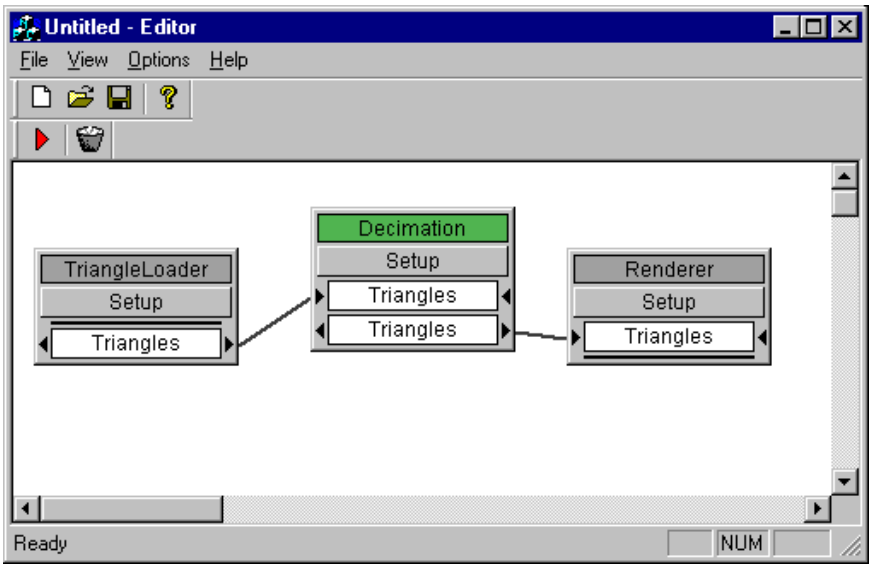


Figure 3.8: Visualisation of triangles with decimation

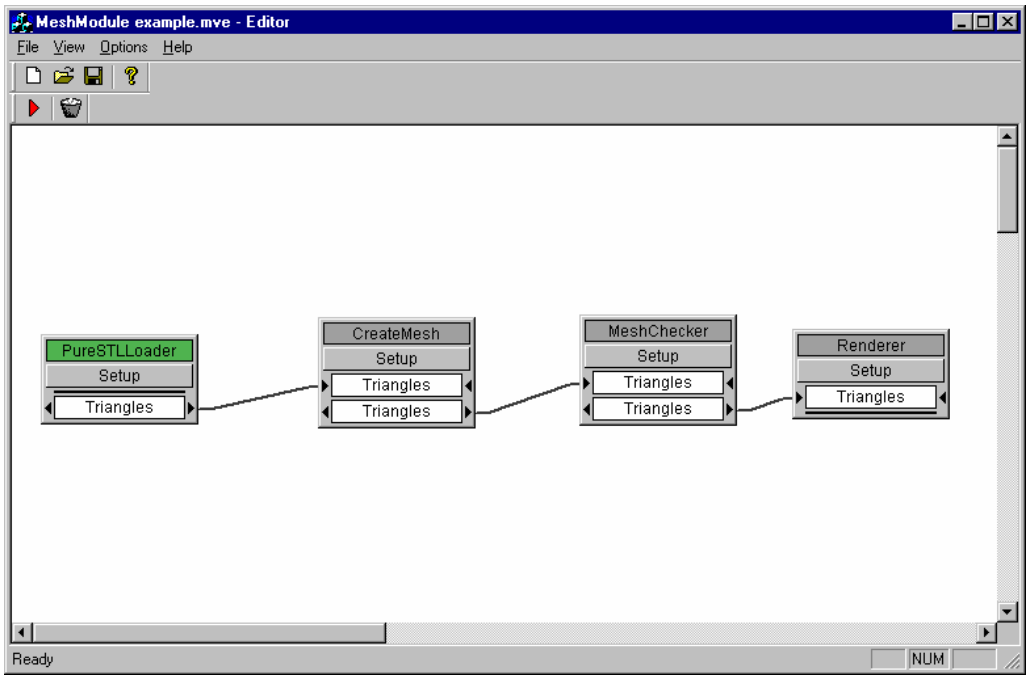


Figure 3.9: Adding useful information into triangles

3.4.3. Visualization and operations over volumetric data

Volumetric data (usualy medical or mechanical CT,MRI) contains objects which can be visualized in various ways.

The first way is by the extraction of isosurfaces (triangles). A scheme for this type of visualization is shown in figure 3.10. Another way of visualizing data of this type is raytracing, which is still under development.

In figure 3.11 there is a scheme describing how to use discrete transformations on volumetric data.

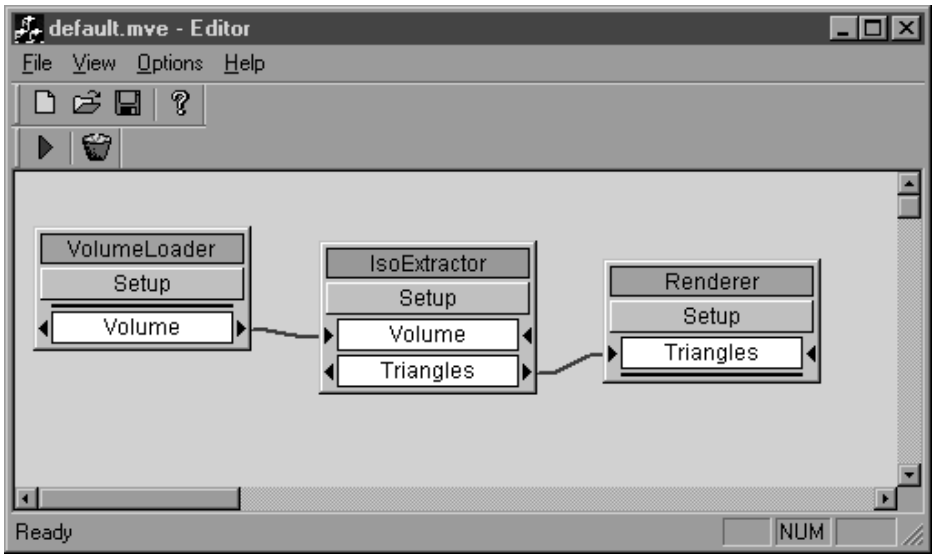


Figure 3.10: Visualization of volumetric data using isoextraction

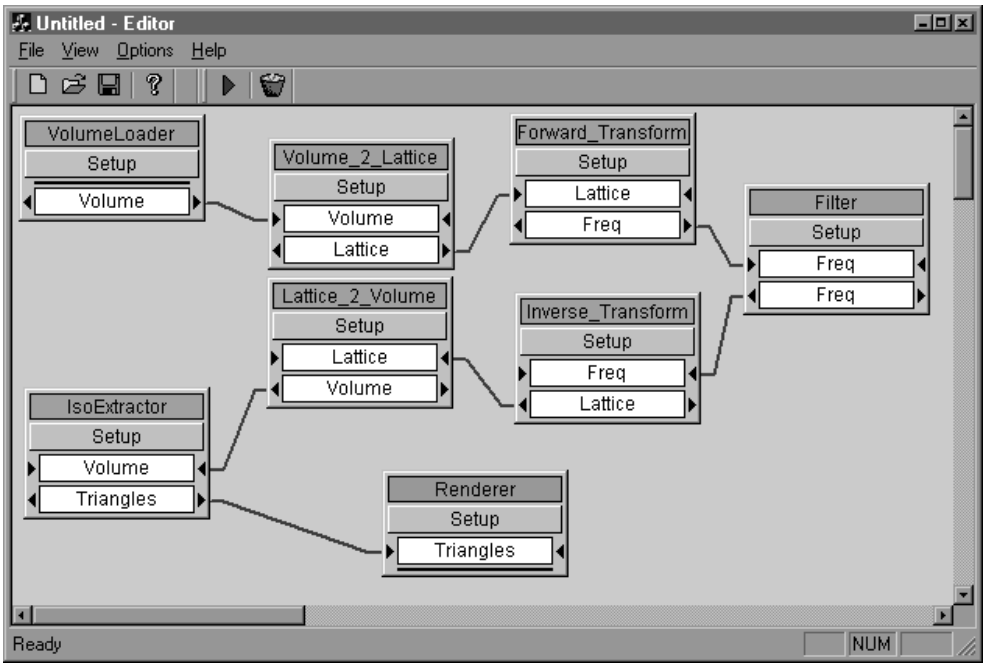


Figure 3.11: Visualisation of volumetric data with discrete transformation

3.4.4. Visualization of slices

Right now there is only one way for getting slices and that is from a triangle mesh. Also you can get a triangle mesh from slices. Slices can be separately displayed. In figure 3.12 there is such a scheme.

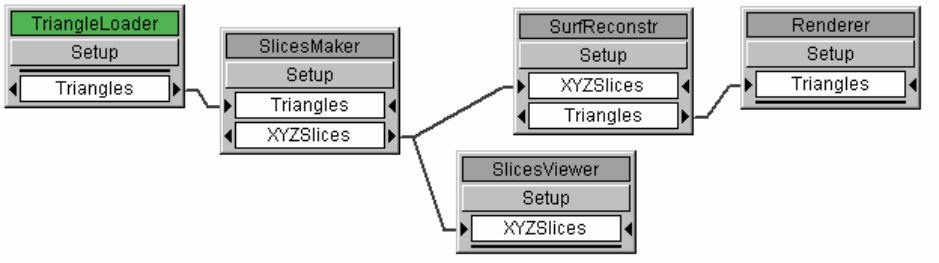


Figure 3.12: Visualisation of slices

3.4.5. List of example data

These are sample files that where used for the examples with the MVE. They are also available at the Internet, on the following locations:

Type		Name	Size	Courtesy
Tri		Bell.tri	28,776kB	
		bone.tri	9,052kB	Cyberware
		bunny.tri	9,467kB	Gatech
		cow.tri	682kB	Avalon
		hand.tri	87,772kB	Gatech
Stl	Bin	Dino50.stl	2,750kB	Cyberware
		teapot.stl	7,794kB	
		teeth25.stl	2,849kB	Cyberware
Stl	Ascii	Baba.stl	9,287kB	Cyberware
		isis25.stl	20,381kB	Cyberware
Vol	1byte	bentum.vol	16,385kB	
		cthead.vol	7,233kB	University of North Carolina
		ctmayo.vol	2,049kB	
		engine.vol	7,041kB	
		hplogo.vol	2,213kB	
		syn_64.vol	257kB	
Vol	2byte	Cthead.vol	14,464kB	University of North Carolina

Cyberware : http://www.cyberware.com

Gatech : http://www.cc.gatech.edu/projects/large_models/index.html

Chapter 4

The Server

This chapter focuses on the server aspect of the application, and the design, implementation, and interaction during its runtime with the MVE and the client.

4.1. About the Server

Most of the applications that use a network to communicate with other applications are developed in the form of a client-server architecture. This means that there is a part of the application that acts as a server, which listens to a network socket – usually a TCP socket – and accepts connections from clients, and there is another part of the application, the client, which asks to connect to the server, in order to exchange messages that are part of the runtime of the application.

In recent implementations we may encounter a different approach, as peer-to-peer communications are gaining ground on applications with increased network traffic, and therefore, every unit acts as a client and a server at the same time, and communicates with its peers, who act the same as itself. This way, the network traffic is reduced, since the messages are exchanged through interested clients, and there is no localized server to handle all requests.

The implementation may vary from application to application, but the general idea remains about the same.

However, in our case, the messages exchanged are short in length, and relatively few during the runtime, thus we adopted the classic client-server architecture for our implementation.

4.2. Interaction between the MVE and the Server

The server in our implementation is a JAVA console application, which accepts connections from the clients, and propagates their requests to the MVE. In order to achieve remote access to MVE, the environment had to be enhanced to accept command line arguments. We did this so that the MVE could be called from an external application, in our case the server, so that it could load a scheme and execute it without opening the editor window. The command line arguments consist of some parameters that define the filenames to be used as input instead of the user interaction.

The basic idea of MVE Remote execution is to provide the client with a list of the available modules, to load and execute a scheme given by the user (the scheme may

consist of the default modules provided to the system, or other modules, for which the user has provided the containing .dll), and to provide statistical information on the runtime in the form of a log file.
In order to acquire the requested list of available modules, the application examines the .dll's located in specific directories defined by the system (like a Modules directory) or the user (like a file containing the modules' path). We can accomplish this by providing the following parameter in the command line arguments:

```
MVE_Remote.exe  -lm  [file name]
```

The argument -lm indicates that the user is requesting a list of the currently available modules. If this parameter is followed by a file name, the filename is the output file in which the list of modules is written. If no filename is given, then the default – Modules.xml – is used.

If a user wishes to execute a scheme, he has to provide the file containing his scheme, and if the scheme contains modules that were implemented by the user and are not yet available to the system, they must be also specified. The command line arguments for MVE are:

```
MVE_Remote.exe  -ex  <file name>  [-m  <file name>]
```

The parameter -ex indicates that a scheme should be loaded into MVE and executed. The name following this parameter specifies the filename that contains the scheme. In case the scheme contains one or more modules that are unknown to the system, the -m parameter is used to specify these modules. The file specified by the following filename contains the .dll's of these modules.

In case anything goes wrong during the process, we should be able to trace the steps taken to see what went wrong. Thus, we want a log file to be generated as a precaution. In order for a log file to be kept, the command line is as follows:

```
MVE_Remote.exe  -lm  [file name]  -log  [file name]
```

The argument -log indicates that a log file should be kept in the file ‘filename’. Obviously, there is no point in giving this argument on its own, since no action would be taken from MVE. The -log argument is used in conjunction to another parameter. Note that the log file is always kept by the system, even if this argument is omitted. The log file is then kept in the file MVE_Remote_Log.xml. This means that we would only use the -log parameter in case we needed a log with a different file name.

A concise list of the available parameters and some examples, are shown in the figure below.

Parameters for execution:

```
MVE_Remote.exe -ex <file name> [-m <file name>] [-log [file name]]
MVE_Remote.exe -lm [file name] [-log [file name]]
```

`-lm [file name]` - File name to write the list of modules that were found in MODULES directory. If no file name is specified Modules.xml is used.

`-ex <file name>` - Name of the XML file with the scheme for execution.

`-m <file name>` - File name to be used to get Module DLL names for execution instead of scanning MODULES directory. Can be used only with -ex parameter!

`-log [file name]` - File name of the log file. If not specified, the default is used "MVE_Remote_Log.xml"

Examples of parameters:

```
MVE_Remote.exe -lm
```

- Writes Modules.xml file with a list of all available modules and the log is in the MVE_Remote_Log.xml file.

```
MVE_Remote.exe -lm My_List.xml -log My_Log.xml
```

- Writes My_List.xml file with a list of all available modules and the log is in the My_Log.xml file.

```
MVE_Remote.exe -ex My_Scheme.xml
```

- Executes a scheme in the file My_Scheme.xml, scans the /Modules directory for the list of modules and keeps a log in the MVE_Remote_Log.xml file.

```
MVE_Remote.exe -ex My_Scheme.xml -m Modules.xml
```

- Executes a scheme in the file My_Scheme.xml, reads the modules' list from the Modules.xml file and keeps a log in the MVE_Remote_Log.xml file.

```
MVE_Remote.exe -ex My_Scheme.xml -m Modules.xml -log My_Log.xml
```

- Executes a scheme in the file My_Scheme.xml, reads the modules' list from the Modules.xml file and keeps a log in the My_Log.xml file.

In order for the Modular Visualization Environment to manage the command line arguments, a number of additional classes were written as part of the package. We mention their names and descriptions:

class CModuleStore : public CObject

- This class is responsible for loading the list of modules from the MODULES directory or XML file that was previously produced by MVE_Remote.

class CMVELogger : public CObject

- A class for storing a complete log and writing it to an XML file

class CXMLSchemeLoader : public CObject

- This class was created for loading schemes from an XML file

class SAXMVEBaseHandlers : public HandlerBase

- A class that just handles some base error messages

class SAXModulesHandlers : public SAXMVEBaseHandlers

- A class for parsing an XML file that contains the list of modules

class SAXSchemeHandlers : public SAXMVEBaseHandlers

-A class for parsing an XML file that contains the scheme for execution

4.3. Interaction between the Server and the client

The Server of our implementation interacts with the MVE through system calls, since the MVE does not support a socket interface. This means that the server and the MVE must be located on the same physical machine, in order to communicate with each other. On the other hand, our aim is to be able to have remote access to the MVE, and if we are dispersed, we want to have access to it through a socket interface. That is the reason why the server was necessary in the first place. The client aspect of the implementation is a web-based application, in other words a JAVA applet, which means that the client wants to communicate with the server through a socket, making the existence of the server necessary.
The server communicates with the client in order to retrieve the schemes the clients sent to the MVE, and calls MVE with these schemes as arguments. It is also responsible for saving and loading the clients' schemes – to a storage medium located on the server's machine – and allowing the collaboration between the users.

The server and the client do not need to have continuous communication, so the socket is utilized for only a fragment of the session. Furthermore, the amount of data that is transmitted during a session is so small, that the network status rarely affects the runtime performance.

Let us now see what information is exchanged between the client and the server during a session. The client may request the following:

- Request from the server the currently available list of modules
- Send a scheme to the server for execution, and visualize the results
- Send the scheme to the server in order to save it
- Request a previously saved scheme from the server, to load and process it.

The data transmitted in all cases is in XML format, since XML is rapidly gaining ground in the area of network communications. The files that become arguments to MVE are also in XML format, making things easier for the server, since it does not have to parse the scheme in order to propagate it to the MVE, but it can send it

directly for execution, without any processing. Later on, we will describe the format of the modules list and the scheme in the XML file.

When a client connects to the server, it requests the currently available modules list, so that the user of the client may develop a scheme. The server, immediately after the client connects to it, calls MVE with the -lm argument, and sends the Modules_List.xml file to the client.

The server then waits for the next request from the client. This may be another list of modules request, or a save request, or a load request, or an exec request. Thus, the server listens to the socket, and reads the input. The valid data it could read is one of the strings “MODULES”, “SAVE”, “LOAD”, or “SCHEME”, representing the four cases of request mentioned above, respectively.

If the server reads ‘MODULES’, it calls MVE with the -lm argument, opens the Modules_List.xml file and sends its contents to the client.

If it reads ‘SAVE’, the next data in the socket will be a string representing the filename for the scheme to be saved, and the next data will be the scheme itself (in XML format). This means that it reads the whole data from the socket, then opens a file for writing (with the filename provided by the client), and dumps the data from the socket to the file. The XML Scheme is saved.

If it reads ‘LOAD’, it reads the contents of the directory ‘UserFiles’, sends the filenames to the client, and waits for the next data in the socket. If it is a filename, it opens the file, and dumps its contents to the socket.

If it reads ‘SCHEME’, it reads the next data from the socket, which is the scheme, and dumps the data to the file Scheme.xml. Then it calls MVE with the -ex argument, and if there is data for visualization, on the client’s side, it makes the output files available to the client.

If it reads anything else, it is considered as unrecognized input, and it closes the connection with the client cleanly.

4.4. MVE XML files description

The **MVE_remote** communicates with the server by passing XML files. When a new client connects to the server, the server will ask MVE for the list of available modules. MVE is called with parameter –lm. As a response MVE will produce an XML file called modules.xml, where all the currently available modules are described. The file is defined according to modules.dtd file. The structure of both files is following:

Modules.dtd

```
<?xml encoding="ISO-8859-2"?>

<!ELEMENT MODULES (MODULE*)>
<!ELEMENT MODULE (PARAMETERS, INPUT*, OUTPUT*)>
<!ELEMENT PARAMETERS ANY>
<!ELEMENT PARAM EMPTY>
<!ELEMENT INPUT EMPTY>
<!ELEMENT OUTPUT EMPTY>
<!ENTITY % PAR_TYPE "(NUM | STR | BOOL)">
<!ATTLIST MODULES
      TIME CDATA #REQUIRED>
<!ATTLIST MODULE
      NAME CDATA #REQUIRED
      TYPE CDATA #REQUIRED
      DLL_NAME CDATA #REQUIRED
      DESCR CDATA #REQUIRED>
<!ATTLIST PARAM
      NAME CDATA #REQUIRED
      TYPE %PAR_TYPE; #REQUIRED
      DESCR CDATA #REQUIRED
      MANDATORY (TRUE | FALSE) #REQUIRED>
<!ATTLIST INPUT
      TYPE CDATA #REQUIRED
      DESCR CDATA #REQUIRED>
<!ATTLIST OUTPUT
      TYPE CDATA #REQUIRED
      DESCR CDATA #REQUIRED>
```

As it is clear from the syntax of the modules.dtd file, there are supposed to be tags called MODULES, MODULE, PARAMETERS, PARAM, INPUT and OUTPUT. The MODULES tag is a root tag and therefore it must be present in the XML file. The only attribute in this tag is that of the TIME, which shows when the list of modules has been created. Each module is described in a separate tag called MODULE. This tag has some attributes and also requires some other tags. The attributes are: NAME, which specifies the module name, TYPE, which defines if the module is either loader [1], computational module [2], saver [3] or renderer [10] (the type is specified by the

number, see Table 4.1). The DLL_NAME attribute is not useful on client side, but it will save the MVE some time when it processes a scheme that was sent by the user. This attribute describes the relative path according to the MVE, where the module is actually stored on the server. The last attribute is called DESCR and gives the client a brief description about module's function.

As it has been already mentioned, there are also some tags required. The tag that is always present (even if it has an empty body) is called PARAMETERS. This tag has no attributes but can contain another tag PARAM, which describes the possible module parameters. The PARAM tag is a so called empty tag (it has no body), but it has four attributes: NAME, TYPE, DESCR and MANDATORY. The NAME attribute specifies the name of the parameter, TYPE is the data type of the parameter [NUM|STR|BOOL], DESCR is a brief description of the parameter and MANDATORY tells if the user is obligated to set this parameter or not [TRUE|FALSE]. Each module can have none or several such parameters.

According to the module function there are also specified tags describing its inputs and outputs. The INPUT tag is an empty tag with two attributes, TYPE and DESCRIPTION, where TYPE is the data type (see Table2) and DESC is a brief description of the INPUT. The OUTPUT tag is similar to the INPUT tag.

Here is the example of the list of modules:

Modules.XML

```
<?xml version="1.0" encoding="iso-8859-2" standalone="no"?>
<!DOCTYPE MODULES SYSTEM "modules.dtd">

<MODULES TIME="Tue Jun 26 18:48:54.471 2001">
      <MODULE NAME="Decimation" TYPE="2" DLL_NAME="Modules\
           Decim_module.dll" DESCR="Triangle Mesh Decimation Module">
            <PARAMETERS>
                  <PARAM NAME="Reduction" TYPE="NUM" DESCR="Percent
                              of reduction"  MANDATORY="FALSE"/>
            </PARAMETERS>
            <INPUT TYPE="2" DESCR=""/>
            <OUTPUT TYPE="2" DESCR="Output : Reduced list of
                     vertices, reduced list of triangles"/>
      </MODULE>
      <MODULE NAME="TriangleSaver" TYPE="3" DLL_NAME="Modules\
                Triangle_Modules.dll" DESCR="Save Triangle Data">
            <PARAMETERS>
```

```
                  <PARAM NAME="FileName" TYPE="STR" DESCR="Name of
                       file to save triangles (*.stl)"
                       MANDATORY="TRUE"/>
                  <PARAM NAME="Binary" TYPE="BOOL" DESCR="Binary file
                       ?"  MANDATORY="FALSE"/>
            </PARAMETERS>
            <INPUT TYPE="2" DESCR="Triangles to Save"/>
      </MODULE>
      <MODULE NAME="IsoExtractor" TYPE="2" DLL_NAME="Modules\
                  Volume_Modules.dll" DESCR="Compute iso-surface from
                  Volumetric data (IsoExtractor)">
            <PARAMETERS>
            Not available
            </PARAMETERS>
            <INPUT TYPE="5" DESCR="Volumetric Data Input"/>
            <OUTPUT TYPE="2" DESCR="Triangle Data Output"/>
      </MODULE>
</MODULES>
```

When a client wants to start the computation of his scheme, the server will pass to MVE a file with a scheme description (scheme.xml). MVE is called with the parameter –ex <file>, where file is an XML description of the scheme.

The way to build an XML file with a scheme can be found in scheme.dtd file:

```
<?xml encoding="ISO-8859-2"?>

<!ELEMENT SCHEME (OPTIONS?,MODULES,CONNECTIONS)+>
<!ELEMENT MODULES (MODULE+)>
<!ELEMENT CONNECTIONS (CONNECTION+)>
<!ELEMENT MODULE EMPTY>
<!ELEMENT CONNECTION EMPTY>
<!ELEMENT OPTIONS EMPTY>
<!ENTITY % IO_NUMBER "(0 | 1 | 2 | 3 | 4)">
<!ATTLIST OPTIONS PARALLEL (TRUE | FALSE) #REQUIRED>
<!ATTLIST MODULE
      ID ID #REQUIRED
      MOD_TYPE CDATA #REQUIRED
      PARAMETERS CDATA #IMPLIED>
<!ATTLIST CONNECTION
      FROM IDREF #REQUIRED
      TO IDREF #REQUIRED
      INPUT %IO_NUMBER; "0"
      OUTPUT %IO_NUMBER; "0">
```

The root element is called SCHEME and it has no more attributes. Inside the SCHEME there are tags MODULES, with the description of the modules, and CONNECTIONS, which describes how the modules are connected. There is also an empty tag called OPTIONS with an attribute PARALLEL [TRUE|FALSE] that says

to MVE to run either in parallel (multithread) or sequentially. Each module is specified in a MODULE tag. This tag is empty and has just a few attributes: ID, MOD_TYPE, and PARAMETERS. ID is the unique name of the module, and it is the way we can identify modules of the same type. MOD_TYPE is actually the name of the module and this attribute should be equal to the attribute NAME from MODULE tag in the list of modules sent to client (see the previous paragraph – modules.xml). The last attribute is PARAMETERS where all the parameters necessary for the module are stored. The parameters are stored in a long string separated by semicolons. The order of the parameters is given by the order in modules.xml file.
The connection between modules is described in CONNECTIONS tag, where each CONNECTION tag (empty) specifies a connection. The attributes are FROM, TO, INPUT, and OUTPUT. The FROM and TO attributes store the IDs of proper modules and INPUT and OUTPUT is the number of module input or output (in case the module has more than one). The first input (output) has the number 0, which is also a default value for this attribute.

Here is an example of a simple scheme described in a XML file:

```
<?xml version="1.0" encoding="iso-8859-2" standalone="no"?>
<!DOCTYPE SCHEME SYSTEM "scheme.dtd">
<SCHEME>
   <OPTIONS PARALLEL="FALSE"/>
   <MODULES>
      <MODULE ID="TriLoad1" MOD_TYPE="TriangleLoader"
                           PARAMETERS="Cow.tri"/>
      <MODULE ID="Dec1" MOD_TYPE="Decimation" PARAMETERS="80"/>
      <MODULE ID="TriSave1" MOD_TYPE="TriangleSaver"
                           PARAMETERS="Cow2.stl;FALSE"/>
      <MODULE ID="TriSave2" MOD_TYPE="TriangleSaver"
                           PARAMETERS="Cow3.stl;FALSE"/>
   </MODULES>
   <CONNECTIONS>
      <CONNECTION FROM="TriLoad1" OUTPUT="0" TO="Dec1" INPUT="0"/>
      <CONNECTION FROM="TriLoad1" OUTPUT="0" TO="TriSave2"
                                  INPUT="0"/>
      <CONNECTION FROM="Dec1" OUTPUT="0" TO="TriSave1" INPUT="0"/>
   </CONNECTIONS>
</SCHEME>
```

Figure 4.1 shows the same scheme compiled in MVE Editor.

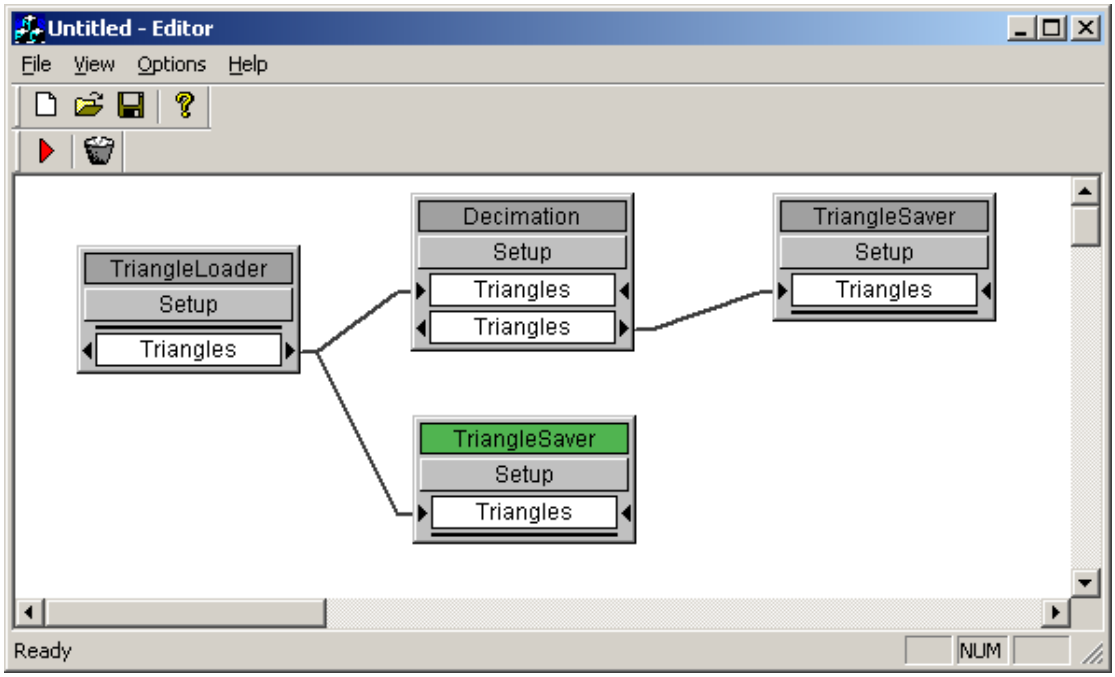


Figure 4.1: MVE Editor with scheme described above

Module type	Number
Loader	1
ComputingModule	2
Saver	3
Renderer	10

Table 4.1: Type of modules according to their function

Data type	Number
Undefined	0
Points	1
Triangles	2
Tetrahedras	3
Lattice	4
Volume	5
Freq	6
FreqHart	7
FreqCos	8
XYZSlices	9
Tetrahedra2	10
Points2	11
Slices	12

Table 4.2: Data types used in description of inputs/outputs of the modules

There is one more optional parameter while calling MVE_remote with some scheme. The parameter is –log <log_filename>. This parameter ensures that MVE will create a log file (XML) with the result of all the operations made during the computation. Since the log file structure is still in progress, there is no final description yet.

Chapter 5

The Client

This part of the implementation is the most essential, since it is the only part the end user gets to use. The other parts of the project are equally essential, but their task is accomplished in the background. This Internet client is the front end for the whole project's implementation, and it has to deal with the users' actions or mishandles.

5.1. About the Client

On every networked application, there is a part of it that acts as a client. Even in peer-to-peer implementations, there may be no server, but there is always a client present. The client is an application that depends on another application for running. A client has to read some data in order to proceed with the execution, and in most cases, if the data it needs is not available, it fails to execute at all.

The amount of data a client needs to exchange with the server or its peers determines the performance of the client, regarding the network workload and traffic etc. If the client has to exchange data often, and in large amounts, the network status may decrease the whole application's performance. Otherwise, if the client's communication with the server or its peers is seldom, and the exchanged data is of short length, the network status does not affect the performance dramatically. However, if the data exchange is necessary for the further execution of the client, the slightest mishap in the communication may result in the total failure of the client.

5.2. JAVA Applets

A client for an application may be of any type of application. With the rapidly increasing use of the Internet, software designers – trying to exploit the capabilities of 'The Net' – tend to use Java Applets for their implementation. The reason for this is that if an application is available on a web browser, it is easier to use and handle (No installation needed – availability on the Internet). Another advantage that Java applets provide, is that they can run on any platform, which means that a Unix, MacOS or OS2 user would run the same applet on their web browser as a Windows user would. Even the architecture of the machine does not matter when running Java classes, so a SUN machine would execute the application in the same way as a PC would.
The GUI (Graphical User Interface) of the applet may be designed so that the user feels that he is working in a Windows environment (buttons, checkboxes, etc), a Unix environment, or a custom environment. The appearance does not affect the

performance of the application, but it is essential to make the user feel that he is working in a familiar environment. The latest Java Distributions support a new set of GUI classes, called SWING, which are considered to be lightweight, and more flexible. However, they are still under development, and no standards have been set for their usage so there might be some problems of compatibility with their use. Our implementation uses the AWT graphics library of Java, which is steady and widely used.

5.3. The Client's GUI

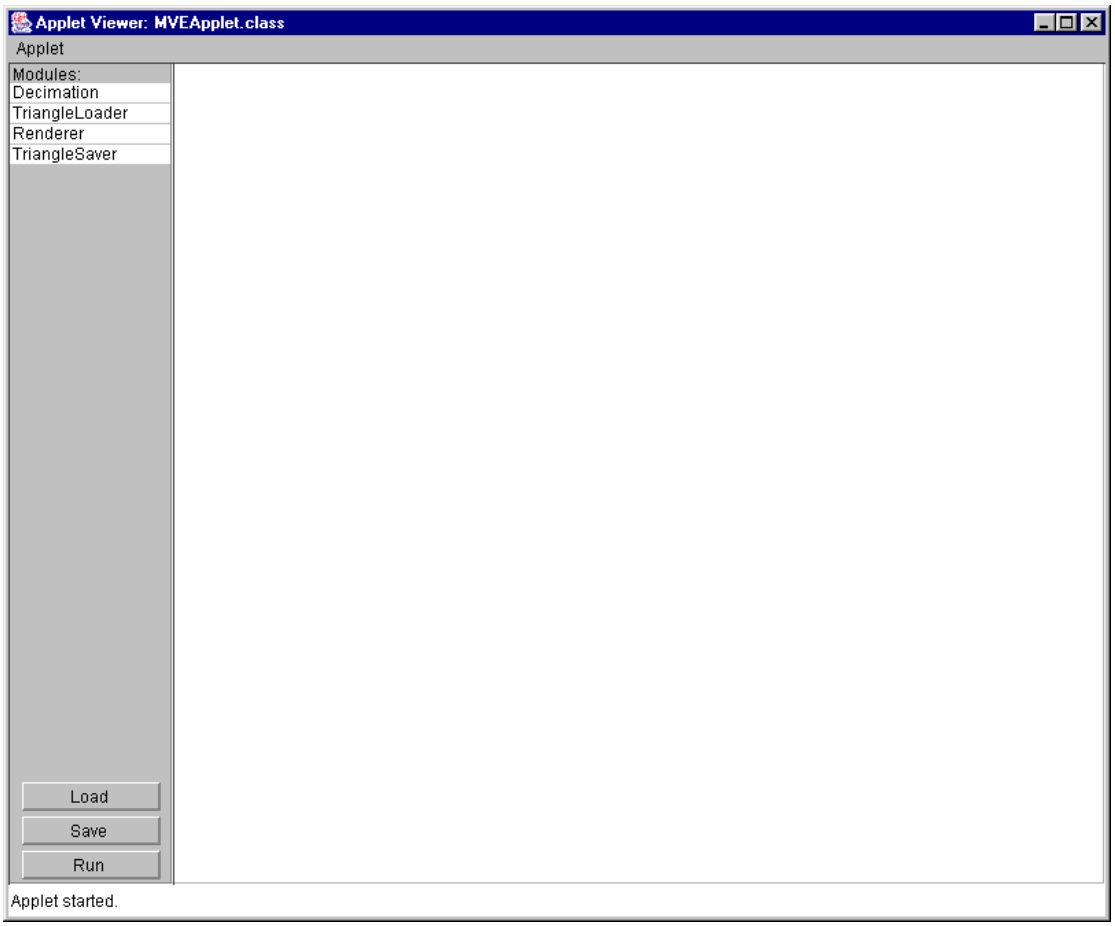


Figure 5.1: The scheme editor of the client-applet (with the list of available modules)

We wanted to make the client look-and-feel exactly like the MVE-Editor, so that a user who has tried out the MVE editor could easily get familiarized with the networked version of the program. Minor changes have been made in the client's

applet, which we thought would suit the user even more than the MVE-Editor. The list of modules on the client is located on a panel on the left of the workspace, and not in another window, available at the menu (View→Modules). The main functions of 'Run', 'Save' and 'Load' are located as buttons on the lower part of the same panel as the modules. The runtime of the applet, however, matches that of the MVE editor in the rest of the details.

5.4. Runtime of the client

When the client is initiated, the first thing it does is to connect to the server. If anything goes wrong with the connection, an exception is thrown and caught and the application has to be reloaded (the GUI is loaded, but no actions could be taken). If the connection is successful, the client requests from the server the currently available list of modules. It sends the string 'MODULES' to the server as the request. As explained previously, when the server reads this string from the socket, it calls MVE with the -lm argument, and sends the output file to the requesting client. The client receives the list of modules in XML format, and has to parse the XML string in order to extract the modules along with their attributes and their parameters. For each module it extracts from the XML string, it allocates a new Object of the class Module, and sets the parameters of the object (input, output, parameters) according to the module. After parsing the whole XML list of modules, we will have an array of module objects, each with its own set of parameters, number and type of inputs and outputs. The new information (the list of modules) is made available to the user, by listing the names of the modules at the top of the panel. If the panel is empty of modules, then the XML file did not contain any modules, or the connection with the server was not established correctly.

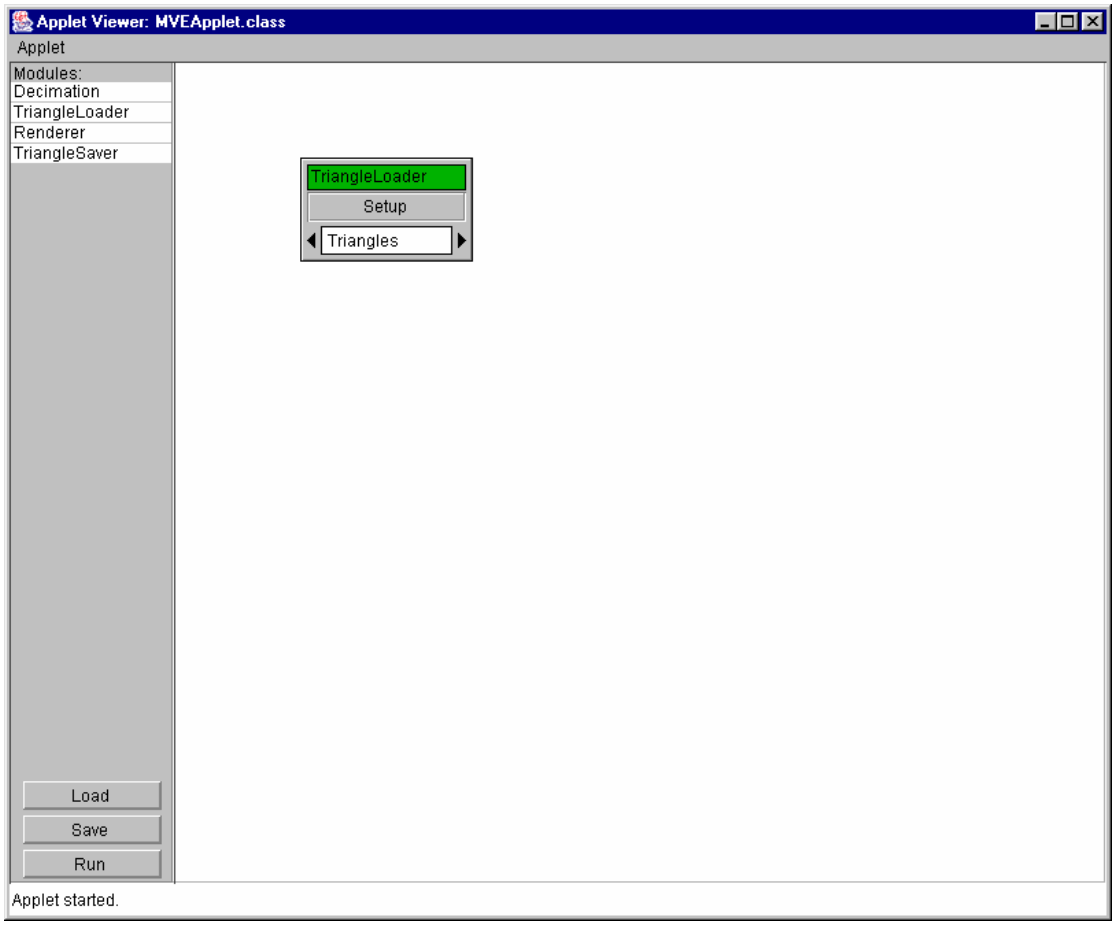


Figure 5.2: Adding modules to the scheme

Once the modules are loaded, the user may start using the workspace as if it were the MVE editor, and develop a scheme. If the user wants to add a module to the scheme, he simply drags with the mouse from the module name into the workspace area. When the mouse button is released, a new object is constructed, of the class mdlObject, which is an object of the module previously selected. A small window appears on the workspace, similar to the one on the MVE editor, which illustrates the module name along with its inputs and outputs.

Through the setup button, the user can set the object's parameters. The setup window for each module does not look similar to the respective window on the MVE editor. That is because the only information we have about a module's parameters is the one given in the XML modules list. This list gives only abstract information about the parameters, and so we cannot have the same characteristics for them, as in the MVE. However, all the information we truly need is given, and we can set the parameters in

a general way, as variables of type NUM, STR or BOOL, for number, string and Boolean respectively.

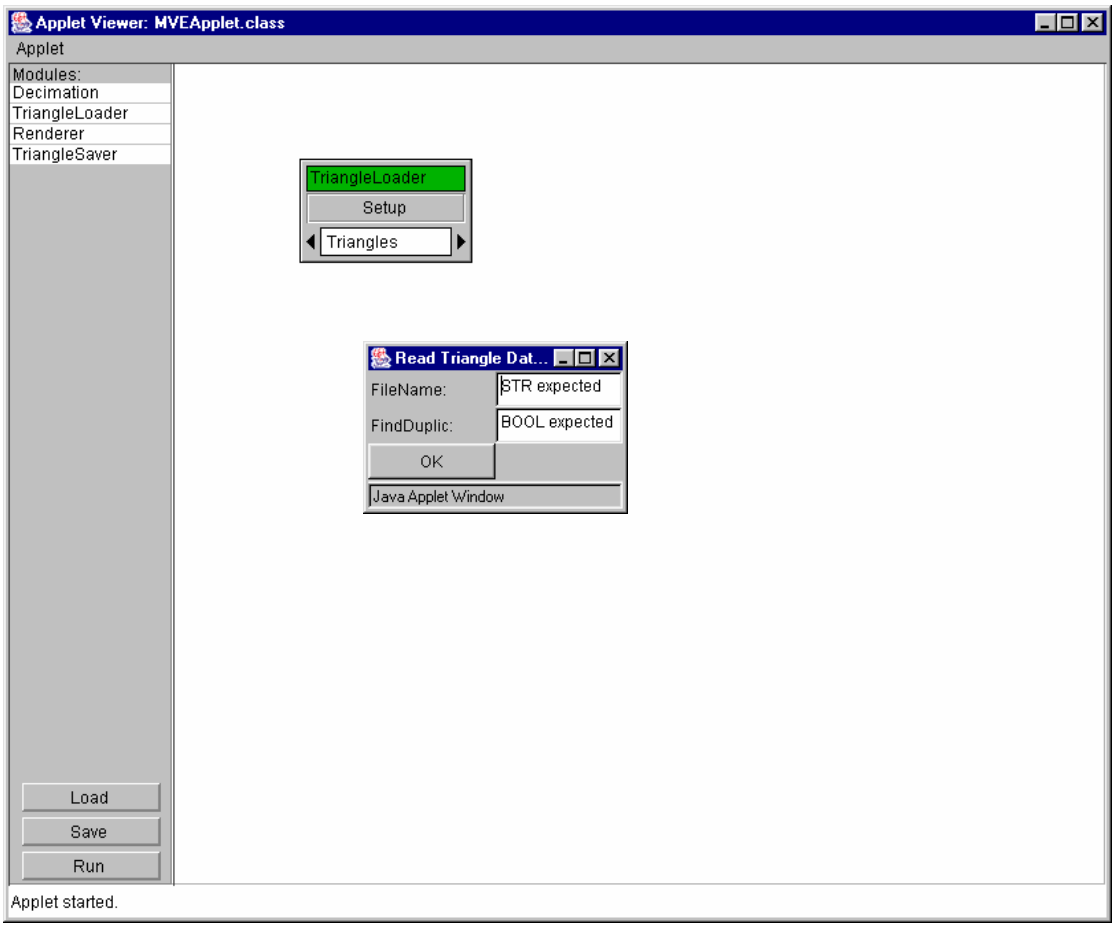


Figure 5.3: Setting the parameters of a module

When the user presses the setup button of the module object, the setup window appears, and the parameters are listed in the form of TextFields, in which the user sets the values. The initial text in each TextField indicates the datatype of the input expected, as shown in figure 5.3. If the value expected is a string (‘STR expected’), any freeform string would satisfy the requirements of the object and would be accepted. If the object expects a boolean value, then only the values ‘true’ and ‘false’ are accepted (written as string of course). In the case of numbers, only digits from 1 to 9 and 0 are accepted. If the user enters an invalid value in the TextField, and presses OK, the window will not close, and the TextField with the invalid value will return to

the initial state, asking for a proper input. When all parameters are set correctly, the window may close, and the values of the parameters are set.

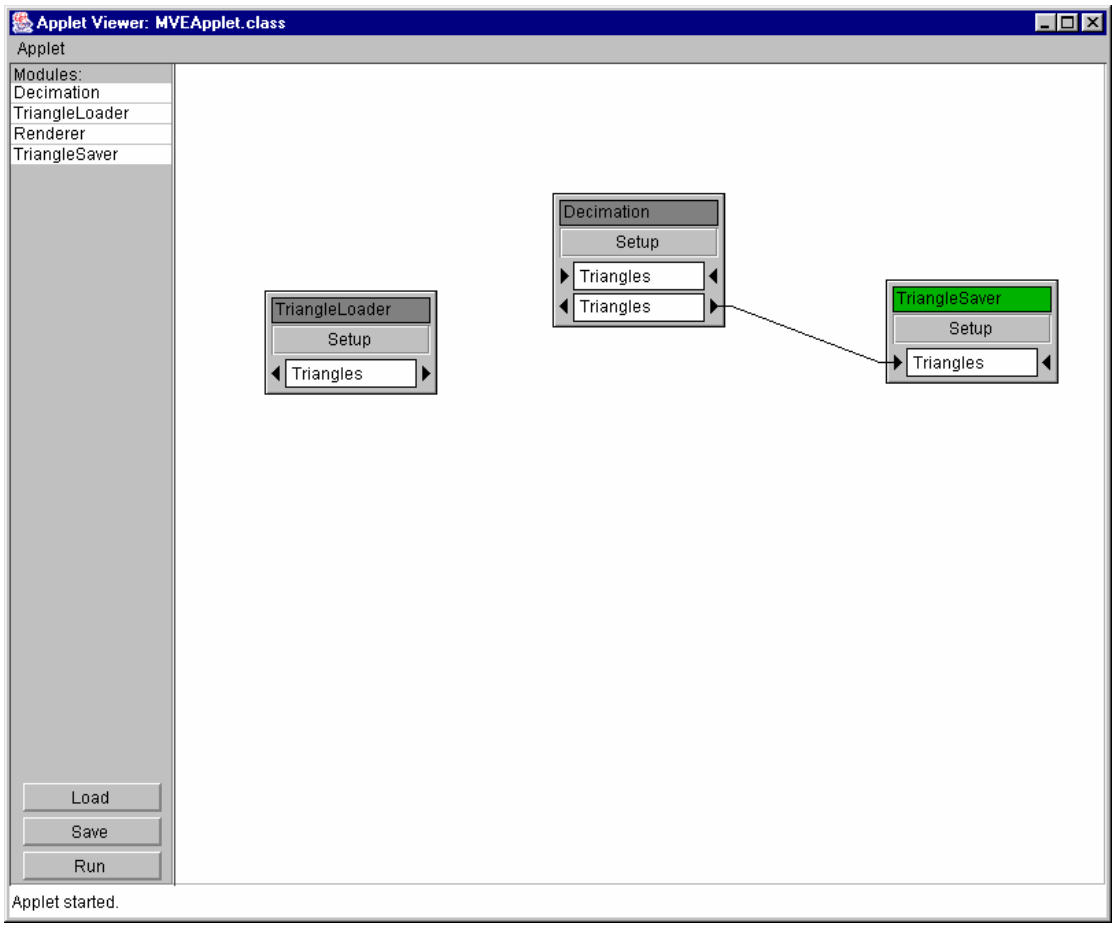


Figure 5.4: Connecting the modules

When the user has finished with the parameters of the module, he can add another module, or many modules, and connect them to each other. The figure 5.4 shows a scheme, consisting of

- a TriangleLoader, which is a module that reads an object from a file – in triangle representation – and loads it to its memory
- a Decimation module, which decreases the number of triangles in the object, and
- a TriangleSaver, which saves the decimated object back to a file.

The parameters of the TriangleLoader are the filename of the triangle representation and a Boolean parameter called ‘FindDuplic’, which searches for unnecessary vertices in the object if set to true. The only parameter of the Decimation module is the percentage of the triangles’ reduction, and the parameters of the TriangleSaver are the filename for the decimated object to be saved, and a Boolean flag for whether the file is binary or not.

The Decimation module and the TriangleSaver module of the scheme in the figure have been connected, so that the output object of the decimation module becomes the input to the Saver module. In order to complete the scheme, the Loader module must be connected to the Decimation module, so that the object that the Loader has read from the file could become input for the Decimation module.

The establishment of a connection is accomplished by dragging the mouse from the output of the first module to the input of the second, exactly like the MVE-Editor. If the type of the input is not of the same type as the output, the connection is not established. If the input has already been connected to another output, the connection is not established. If the connection begins from the output, but does not end at an input, or if the input is of a different type, it is cancelled.

An output may be connected to several inputs, as long as they are of same type. This makes the scheme even more flexible, as we can use one module of some type, which undertakes a task, and the results of this can be propagated to more than one modules, reducing the time of processing.

5.5. Operations on the scheme

At this point, the user has developed his scheme, and to complete his work, he can either save the scheme or run it. In both cases, there has to be some interaction with the server.

5.5.1. Saving a scheme

On the MVE editor, the user may save his scheme simply, in the old fashioned way. He just has to press the 'Save' button, and choose the file to be saved on the Hard Disk. In our case however, it is not quite that simple to save a scheme to a file. Note that the client we are using is web-based application, a Java applet, and it has limited permissions on our computer. What we should do, is to instruct the server to save the scheme for us, on the remote computer of course, where the rest of the files are located – the MVE, the server's classes, the client's classes, and other users' files.

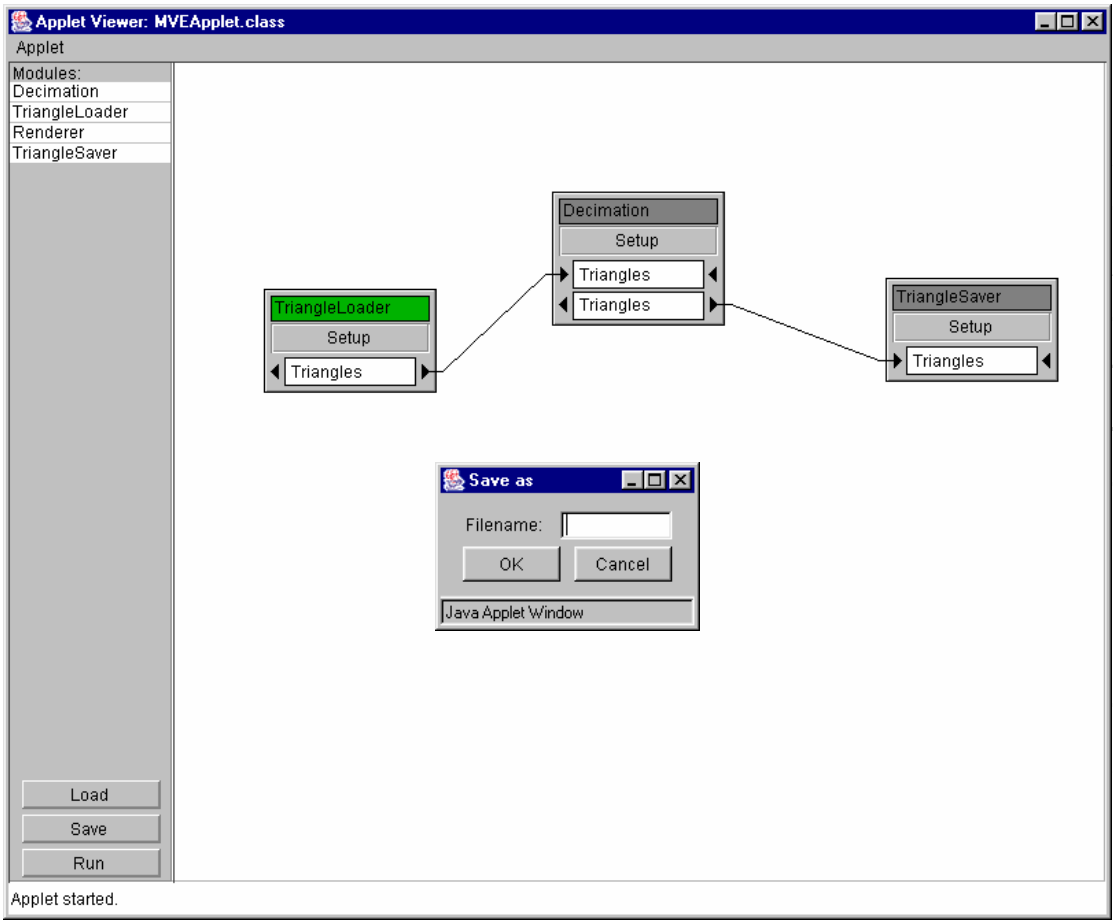


Figure 5.5: Saving a scheme

5.5.2. Loading a scheme

At any point during the session, the user may load a previously saved scheme, in order to execute it again, or for further processing. If any scheme is present on the workspace, unsaved changes at this scheme will be lost, as the workspace is cleared before the new scheme is loaded.

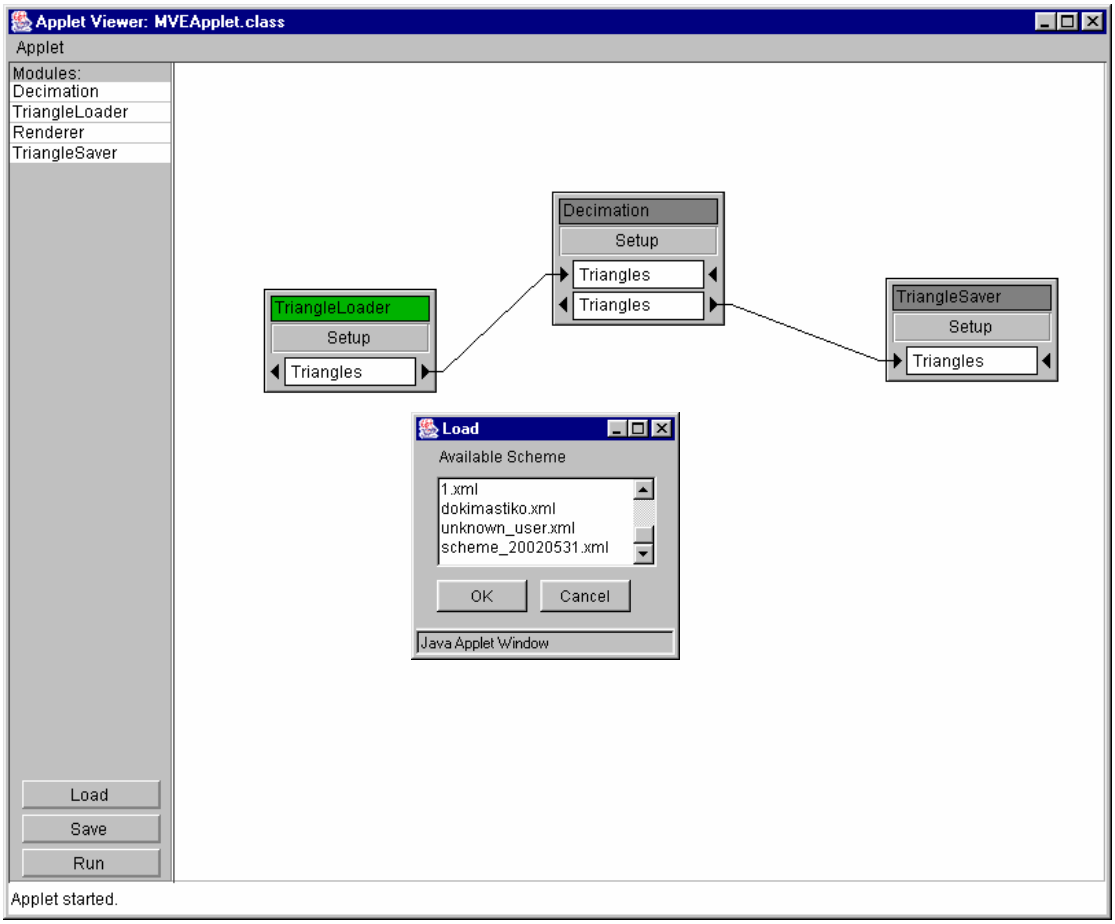


Figure 5.6: Loading a previously saved scheme

To load a scheme, the user presses the ‘Load’ button. The client immediately sends the string ‘LOAD’ to the server. The server, in response, reads the contents of the directory ‘UserFiles’ – where the user’s files are stored – and sends a list of their filenames to the client. Since the client does not know the length of each filename and the number of files, the server has to inform him about the relevant lengths. So the

server writes on the socket the number of files, followed by the length of each filename, and the filename. An example is given at the figure 5.7.

6	
5	1.xml
10	client.xml
15	dokimastiko.xml
10	kokoko.xml
8	test.xml
16	unknown_user.xml

Figure 5.7: The string the server sends to the client in order to extract the filenames that are available on the server's directory

(The numbers represent the length of the list and the items, so that the client could parse the buffer easier)

The client reads from the socket the list of files, and opens a ListBox for the user to choose one of the schemes to load. When the user chooses a scheme and presses OK, the client sends the filename to the server, and waits for the scheme. The server has already read the word 'LOAD', so he is still waiting for a filename. If it reads the filename from the socket, it opens the file and sends its contents to the client. If the remote user has cancelled the operation, instead of the filename, the word 'NULL' is written to the socket, so the server may return to its normal state.

When the client has received the scheme to be loaded – in XML format – he parses the XML string to identify the objects that participate in the scheme, along with their parameters and the connections between them. For every module that is recognized in the XML string, an object of this module is created, and the parameters of the module are set as parameters of the new object. When all objects are created, the client establishes connections between the modules, as instructed in the XML scheme, and the modules and the connections are added to the workspace. The scheme has been loaded.

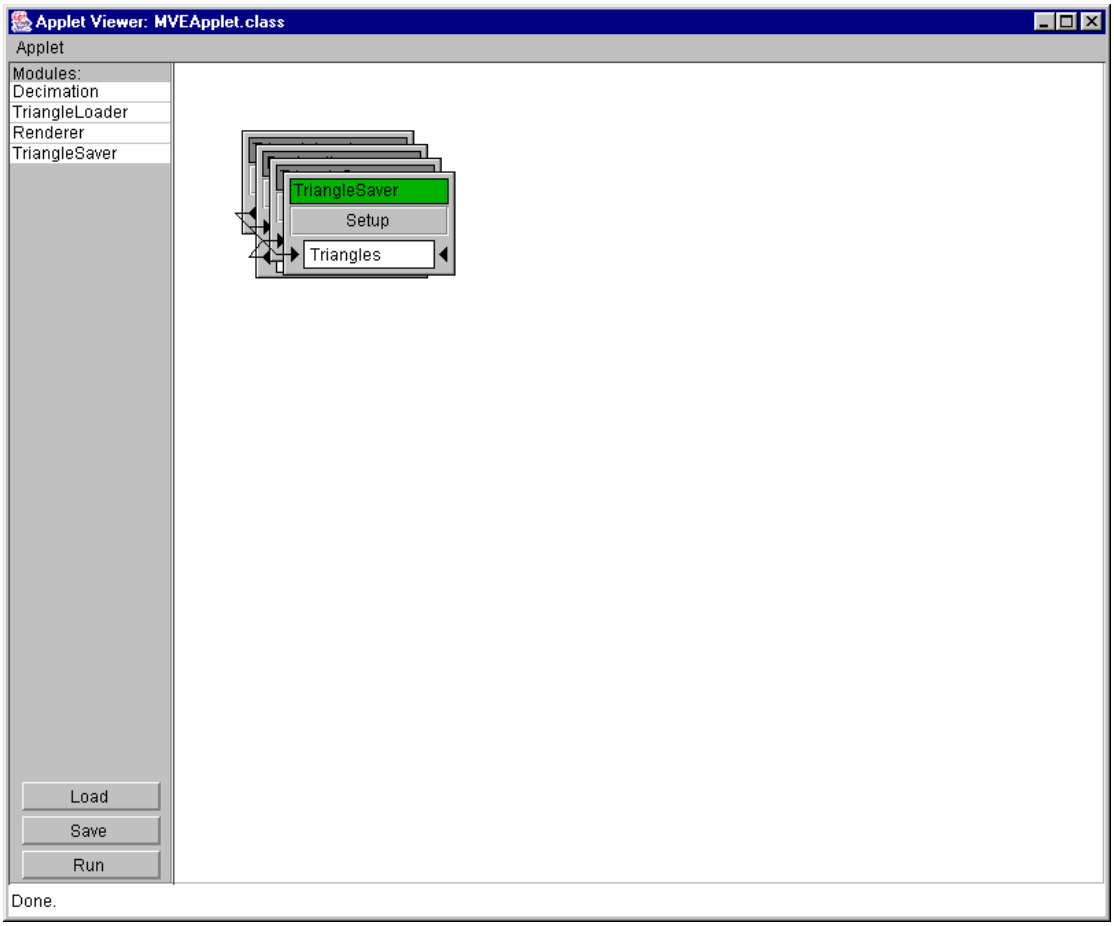


Figure 5.8: A scheme has been loaded. The information about the previous positions of the modules is not absolutely necessary for the scheme, so they are placed in relatively increased coordinates.

5.5.3. Executing a scheme

The third option a user has for a scheme is to run it. The only thing for running the scheme is to press the 'RUN' button. The word 'SCHEME' is sent to the server, followed by the XML scheme. The server writes it to a temporary file, and calls MVE to execute it. If there are any output files, they are stored on the server's directory. If the scheme contained any rendering modules, the output files with the image (or VRML object) are sent to the client. The client renders the resulting objects. The example illustrated in the figures is as follows:

Figure 5.1. We loaded the MVE client
Figure 5.2. We added a set of modules
Figure 5.3. We set their parameters
Figure 5.4. We completed the scheme by establishing the connections between the modules
Figure 5.5. We saved the scheme
Figure 5.6. We loaded another scheme
Figure 5.8. We added some more modules and connections to the loaded scheme
Figure 5.9. We executed the scheme

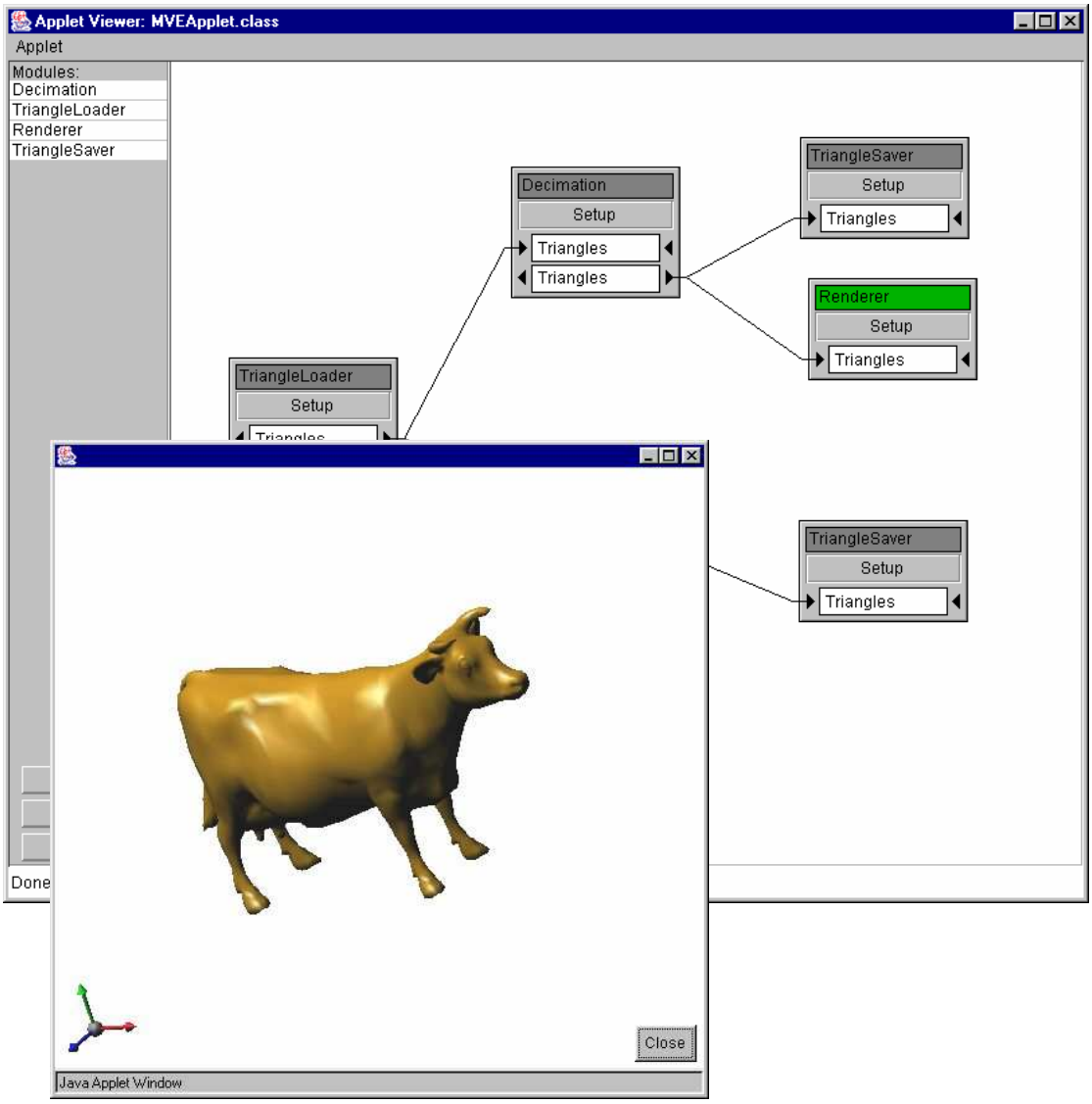


Figure 5.9: The previously loaded scheme has been executed. The window with the cow object is the result of the renderer module

Chapter 6

Experiments – Performance

In this chapter we present some experiments we did on the runtime of our implementation, and the overall performance of our system. As a benchmark we used the Symantec PC-Anywhere tool, which allows remote access to the desktop of the computer. We measured the response times for loading the application and the applet, saving, loading and executing a scheme.

6.1. First Experiment

The following experiment was conducted two times, on two different client computers. One of them had an AMD Athlon 1000MHz processor, 384Mb of RAM and Windows XP as an Operating system. Its connection to the server was established through a 56kbps modem. Its IP address was 62.103.227.97 (a dynamic IP) and from now on we will refer to it as the Dialup Computer. The second client computer was an Intel P3 1000MHz with 256Mb RAM, and the OS was Windows ’98. It was located in the same LAN as the server, so its connection to the server was a 100Mbps Ethernet connection. Its IP address was 195.130.121.25 and from now on we will refer to it as the LAN Computer. The server was an Intel P3 1000MHz, 256Mb RAM and OS Windows ’98. Its IP address is 195.251.195.3.

On the server we installed the Modular Visualization Environment and a test version of MVE_Remote – the Whole environment had to be converted, and the modules as well, so we could only use a fragment of the modules as a sample. The server of our implementation was implemented in Java as a console application. The J2SE 1.4 was installed on the computer also.

The classes of the client are located on the same computer as the server. The reason for this is that the applet could only open a socket to the machine it was downloaded from. This means that in order for the client to establish a socket connection with the server, they both have to be installed on the same machine. This also means that we have to install a web server. As a web server for the client, we installed Apache – it is the most commonly used, and it is free.

On the machines that wanted to execute the client, we only had to install the Java plugin for Iexplorer or Netscape. The rest (eg the network connection) were considered already installed.

To measure the performance of our system, we installed Symantec PC-Anywhere, a tool that exports the whole display of the desktop of one computer to another, so that a user may have remote access as if he had physical access to it.

The next figures show the entire process of a session of MVE via PC-Anywhere, and the same session with our client. The actions taken in each session were exactly the same.

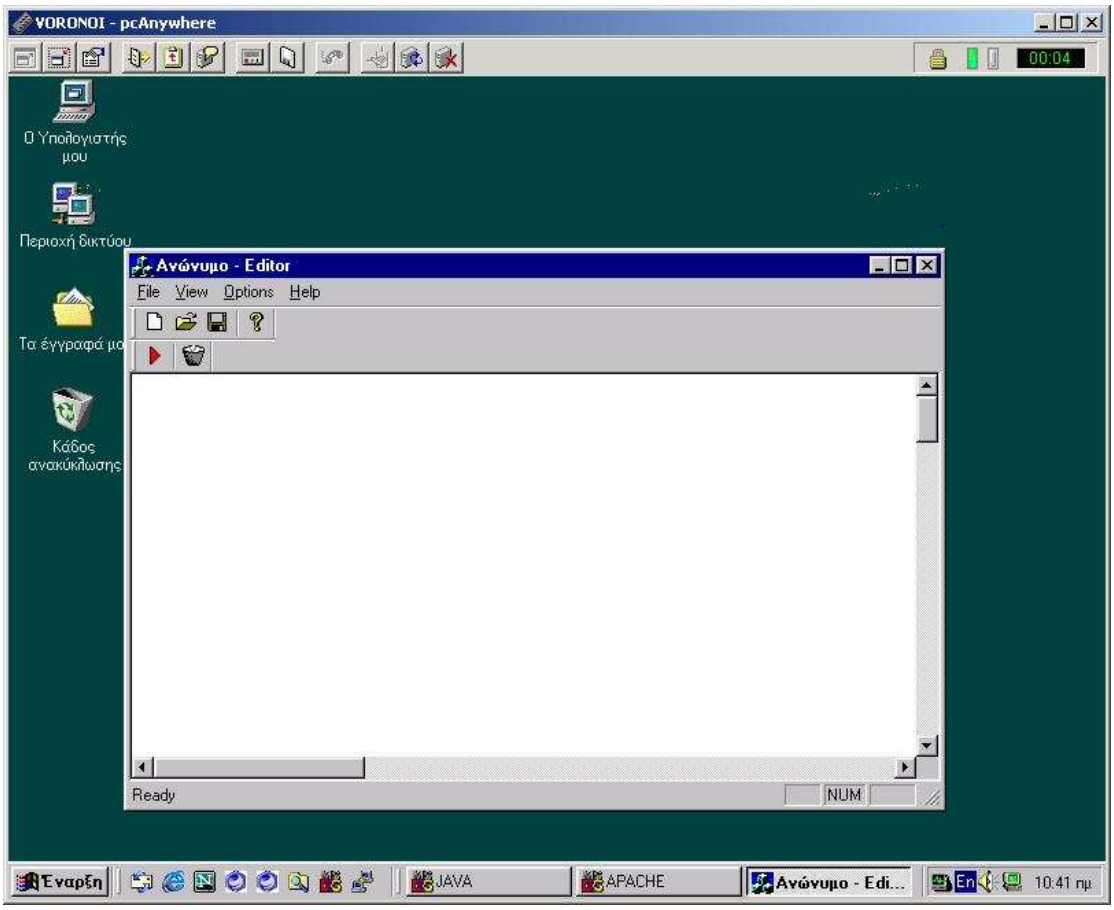


Figure 6.1: PC-Anywhere, the MVE has just been loaded

In this figure we can see the window of PC-Anywhere, containing the whole desktop of the server's machine. At this point we just loaded the MVE, and measured the time it took to load. On the server it took about 1sec to load. On the LAN Computer the time it took to load was 2 seconds, and on the Dialup Computer it took 13 seconds.

At this point we have to mention that with PC-Anywhere tool, the whole session was 'in slow motion', and we had the feeling that the windows and the mouse pointer were moving like inside a tank of liquid! This is because the whole desktop had to be exported to the remote computer. Apart from that, the performance of the session was satisfactory.

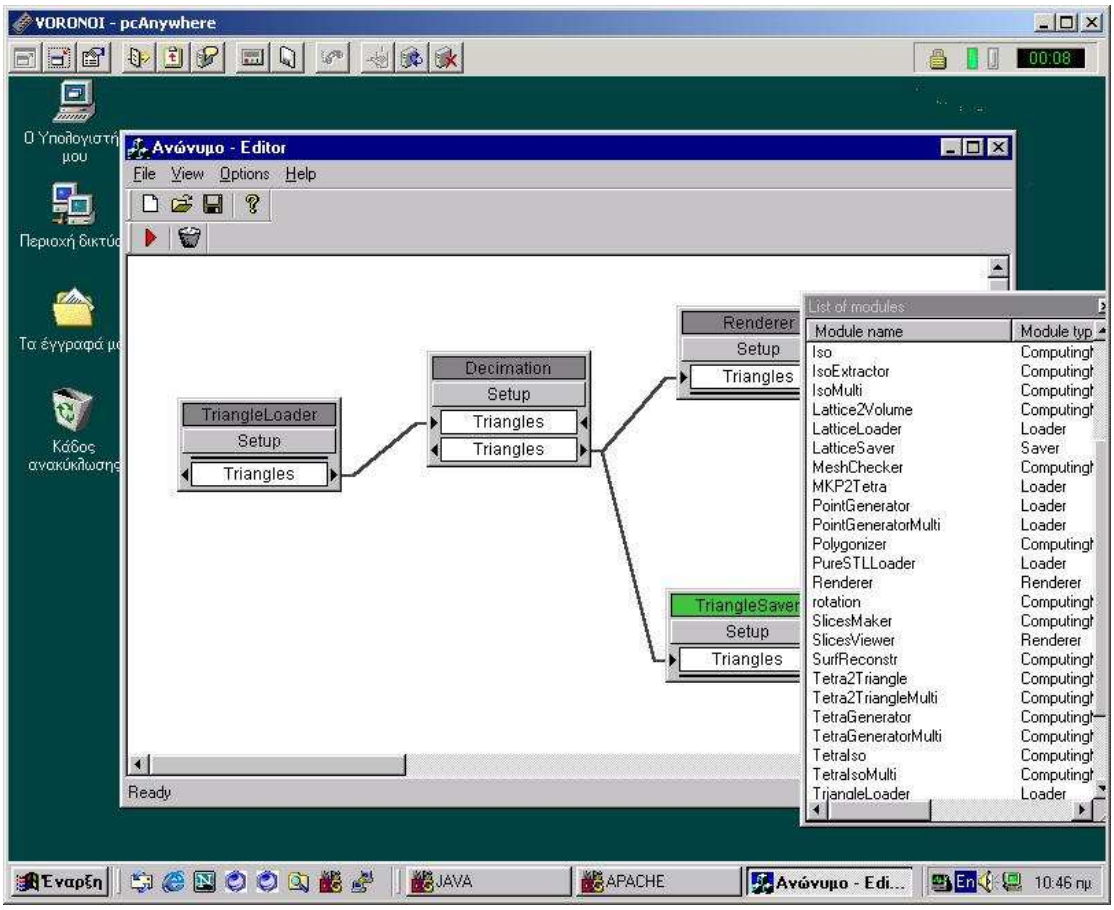


Figure 6.2: The Modules and connections have been set in place

This figure illustrates the scheme we created as a test scheme, and the list of available modules. The scheme consists of four modules, a TriangleLoader, a Decimation module, a TriangleSaver and a Rendering module. The object that is loaded by the loader is decimated, and the resulting object is rendered and saved back to a file. As we can see, the output of one module can become input to more than one modules.

The time to set up the scheme was not measured, because this procedure requires user interaction, and such actions cannot be measured. The actions we measure do not require user interaction, and the time of their completion depends only on the status of the network and the capabilities of the computers involved.

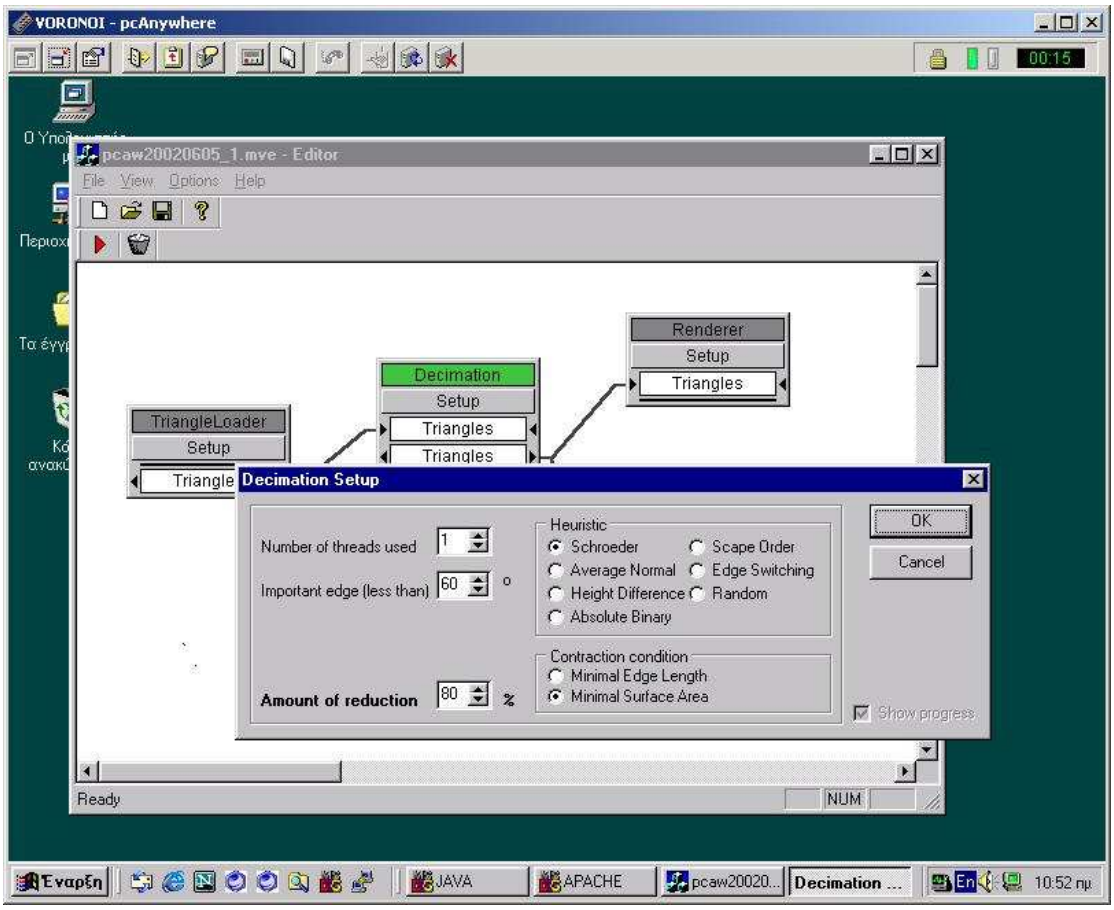


Figure 6.3: Setting the parameters of the modules

Apart from connecting the modules to each other, we must set the parameters of the modules, in order for the scheme to run successfully. In this figure we can see the setup window for the decimation module. This action is also not measured, because the user is the one who sets the parameters.

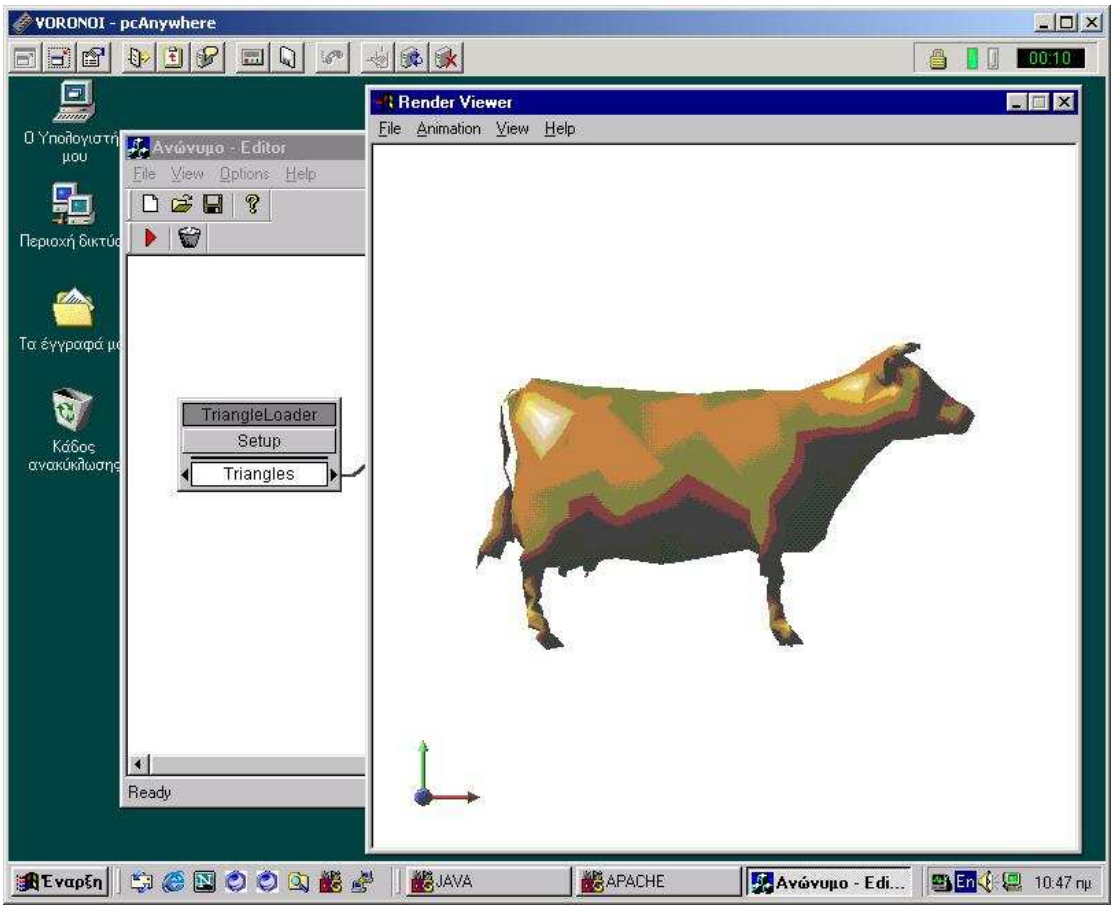


Figure 6.4: Execution of the scheme

When the scheme is completed we can save it or execute it. The window with the cow-model is the result of the Renderer module. The rest of the results – in our case that of the Saver module – are not visible, since all the work on them is made on the server's computer.

This execution of this scheme was measured for our experiments, and it took 6 seconds to execute on the dialup computer, while it took 4 seconds to run on the LAN computer. On the local machine of the server the same scheme would take about 1 second to execute. As we can see, most of the time is lost for the display to be transferred through the network. The actual execution time was not excessive.

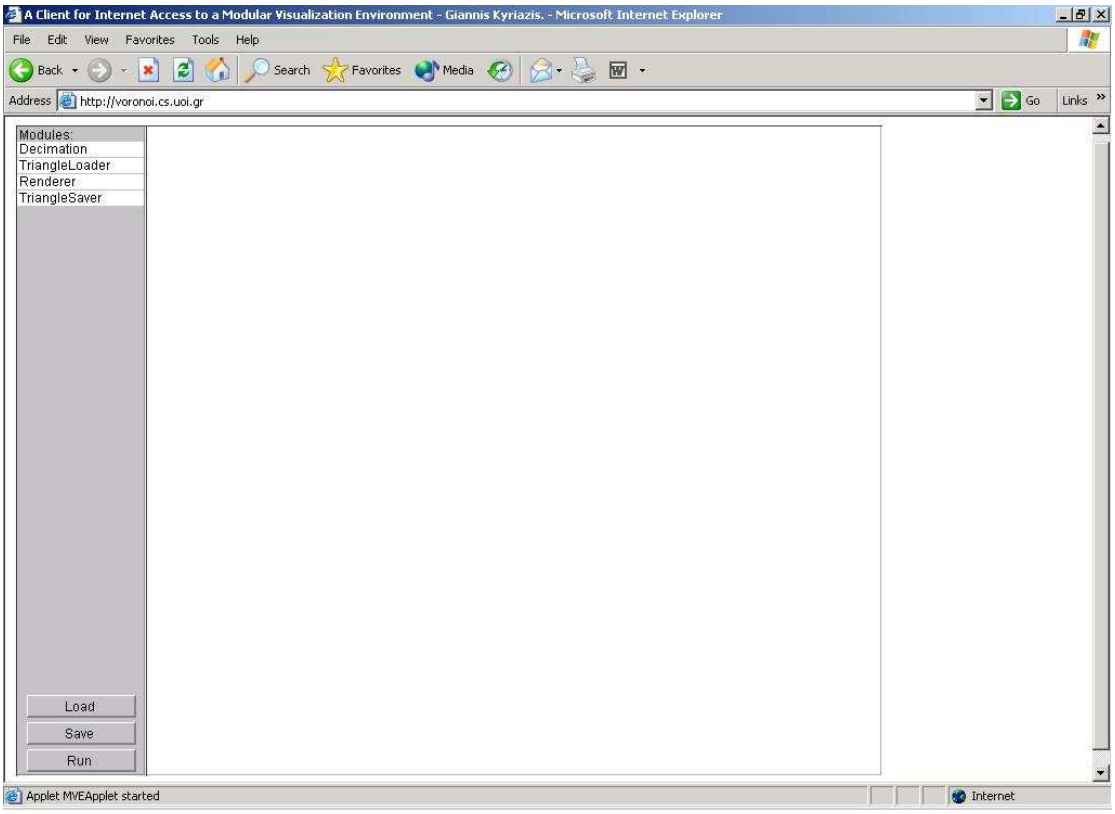


Figure 6.5: The Internet Client has just been loaded

The same session described in the previous figures was repeated using the Internet Client of our implementation. The same machines were used for this session, and the actions taken during the session were generally the same.

In contrast with the PC-Anywhere session, where the whole display had to be transmitted through the network, in the case of the Internet Client the only thing that has to be loaded are the classes of the client's applet. When the client has been loaded, the next thing to do is to get the list of currently available modules. This requires only the transmission of an XML string containing the requested list, which is a small amount of data, relatively to the rest of communications.

The time to load the applet was measured, and it was about 2 seconds for the dialup computer. On the LAN computer and on the local machine where the classes are located the client loaded in about 1.5 seconds. The size of the clients' classes is less than 50kb in size.

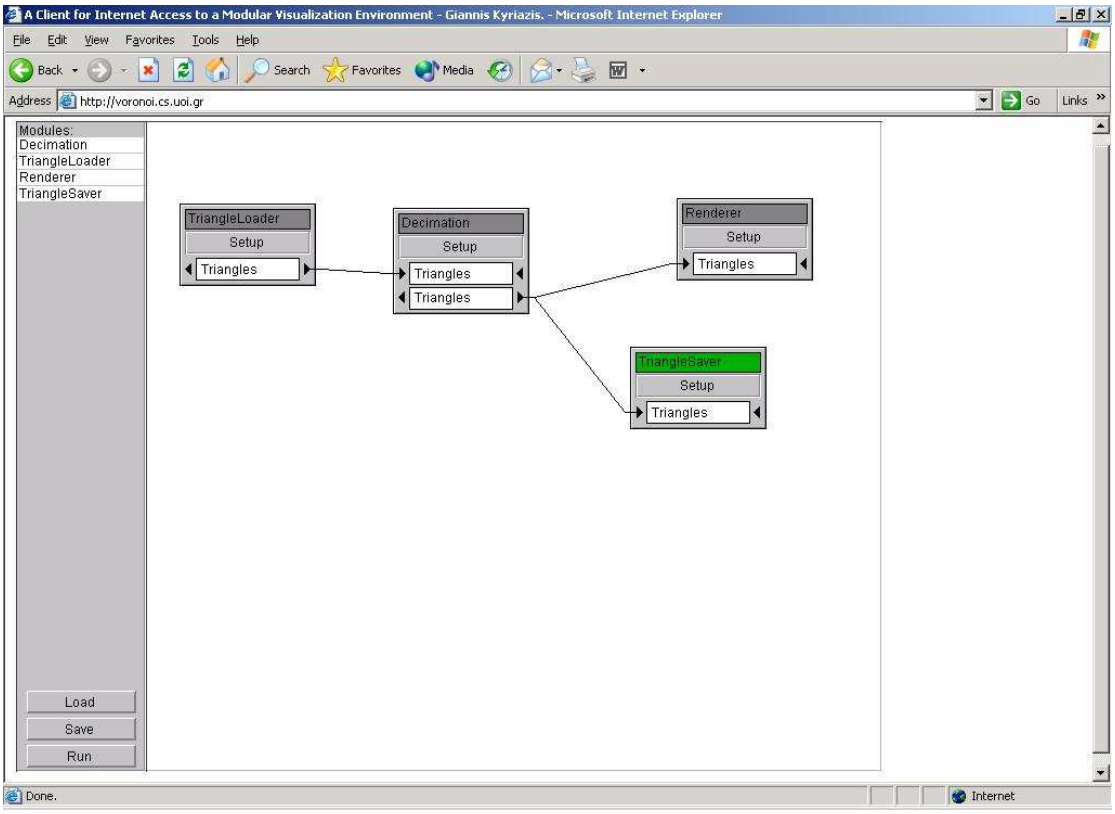


Figure 6.6: Creating a scheme with the Internet Client

We used the same scheme we created on the MVE with PC-Anywhere. There is nothing to mention here, because the process is exactly the same. No measurements were taken, as the process depends on the user's actions.

We should only mention that with the Internet Client we didn't have the problem with the mouse and the windows corresponding to our actions. The problem with the PC-Anywhere session was that the actions done had to be transmitted through the network, but with the Internet Client the only data required to be transmitted through the network is the data, which is absolutely necessary to complete the session.

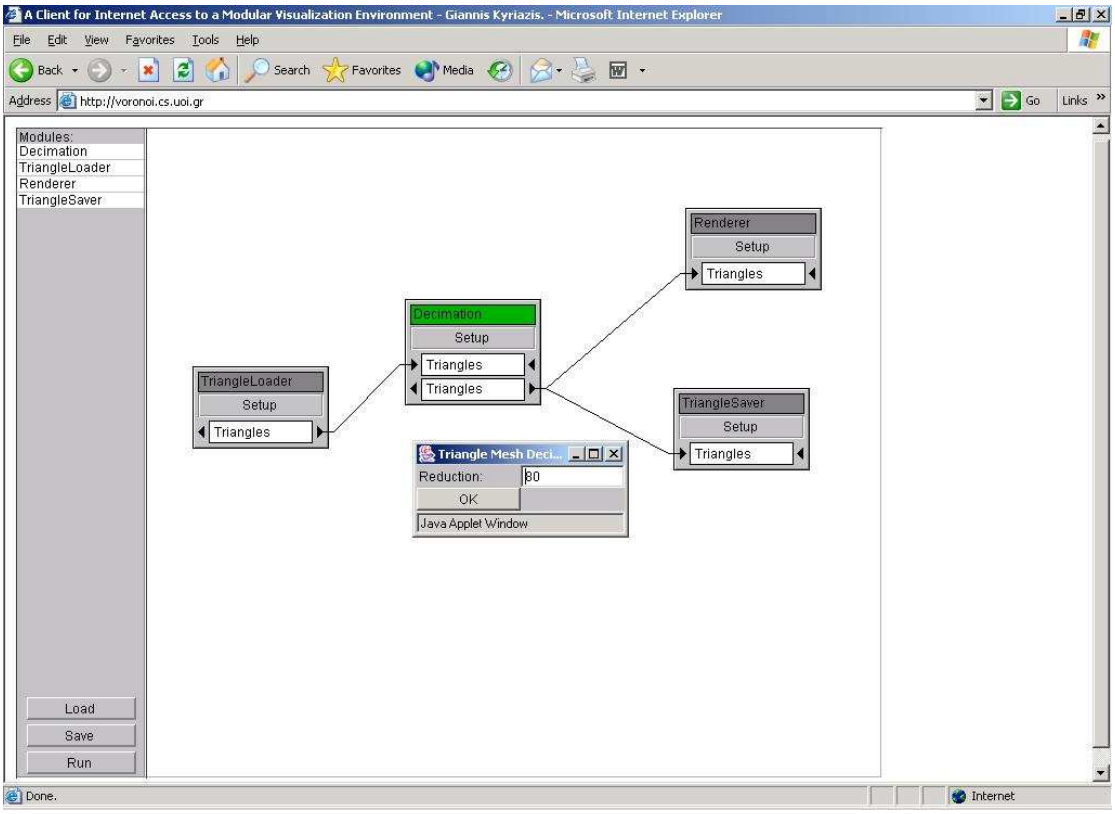


Figure 6.7: Setting the modules' parameters

The figure 6.7 shows the scheme and the setup window of the decimation module, at the time we set the parameters of the modules. As we can see, the setup window is much simpler and less explanatory than the actual setup window of the modules in the MVE. The reason for this is that the only information we have about the modules is what the list of modules could give us. So we cannot provide the user with more information than that which we actually have.

Anyway, our goal is to make a system that could provide remote access to the users with maximum performance. Some features that are considered to be unimportant are ignored, or taken for granted with default values.

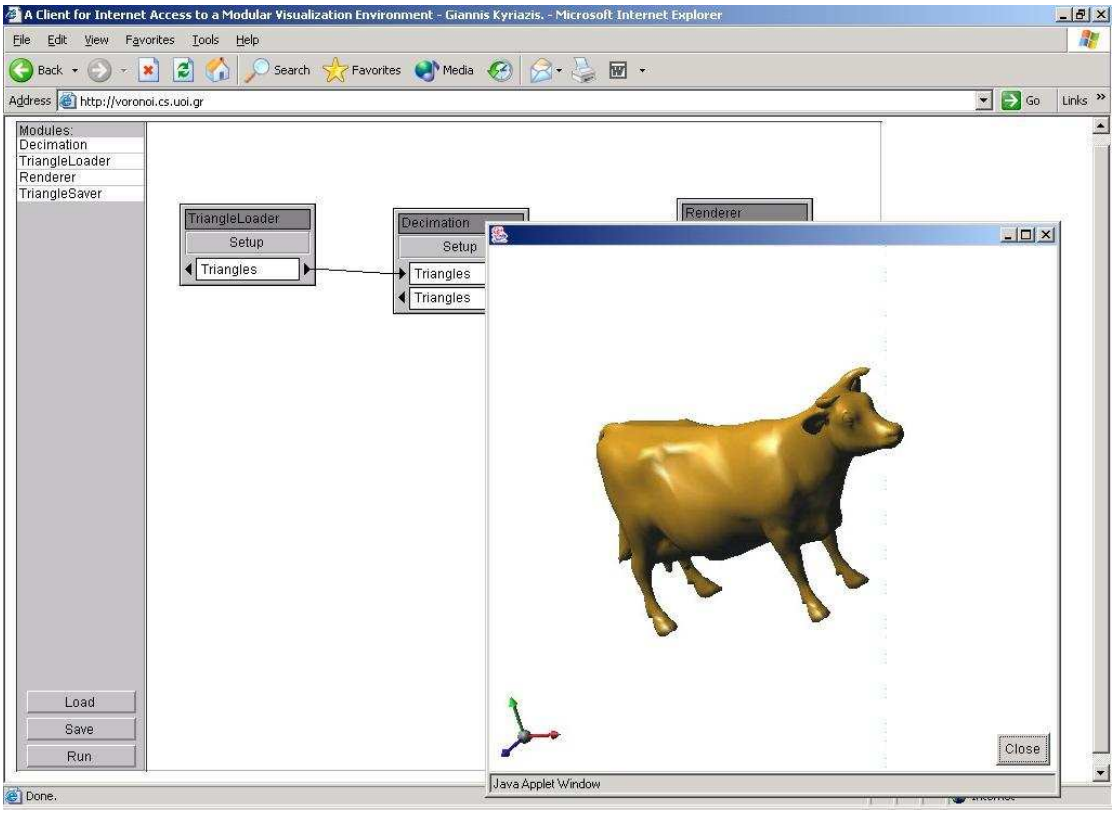


Figure 6.8: The execution of the scheme on the Internet Client

The scheme has been executed on the Internet Client. Again, the only visible result to the user is the cow-model, which is the result from the renderer. If there were no rendering module on the scheme, then the user would have no visual confirmation about the execution of his scheme. Of course, if something went wrong, he would be notified with an appropriate exception that would be thrown and caught. However, in the case of a successful execution, the only visible result is the output of the rendering modules.

The execution time of the scheme was measured, and it took about 1 second to complete on the dialup computer, while it completed a bit ealier on the LAN computer and the computer where the classes are stored (and the server).

The conclusions of our experiments are summarized in the table 5.1:

	Dialup Computer (62.103.227.97)		LAN Computer (195.130.121.25)	
	PC-Anywhere	**Internet Client**	**PC-Anywhere**	**Internet Client**
Time to load	13 sec	2130 ms	2 sec	1650 ms
Time to execute	6 sec	975 ms	4 sec	845 ms
Save Scheme	2 sec	10 ms	About 1 sec	10 ms
Load Scheme	2 sec	280 ms	About 1 sec	130 ms

Table 6.1: Times taken to complete tasks with the Internet Client, and the Benchmark tool

Measurements have also been made about the time it took the system to save and load a scheme. With the pcAnywhere tool the scheme took 2 seconds to save, and 2 seconds to load, when using the dialup computer. On the LAN computer it took about 1 second. With the Internet Client the scheme was saved in 10 milliseconds, on both the remote and the LAN computer. The time to load a scheme on the dialup computer was about 300 milliseconds. On the LAN computer it took about half the time. The measured values are shown in Table 6.1 also.

To conclude, the pcAnywhere tool responded well to the tests we made, and the only problem we had was that there was too much data to transmit over the network. The mouse pointer and the windows where difficult to move, and this was rather annoying. On the contrary, the Internet Client did not require much data to be transmitted over the network, so the overall performance was good. Especially in slow network connections, like a 56k modem, its performance was very good, as it could almost match the LAN connection's performance, and even the performance at the local machine, where the classes and the server are located.

6.2. Second Experiment

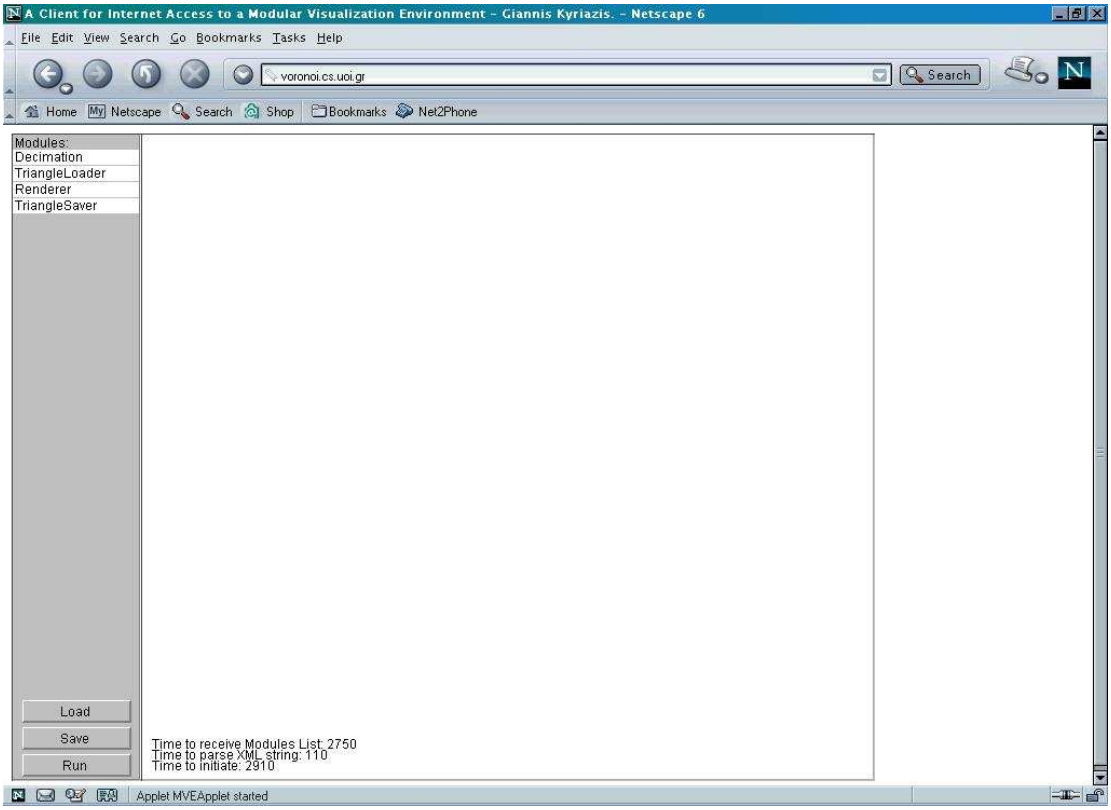


Figure 6.9: The Internet Client took about 3 seconds to load.

We conducted another experiment, in order to measure the performance of our Internet Client, not compared to a benchmark tool, but in various situations concerning the state of the server and the machines where the server and the client run.

We measured the time the client took to load, in a normal state, when other clients had already connected to the server, in situations where there was increased network traffic, and in situations where the server's machine is low on resources.

In order to increase the network traffic, we transferred large files from the server's machine to other machines, during the session of our Internet Client. In order to reduce the resources of the server's machine, we performed heavy tasks on the computer, like encoding video, or applying system checks during the session of the Client.

In order to measure the time it takes for the client to complete a task, we measured the system time (in milliseconds) just before the task began, and immediately after it was completed. The times we measured were the following:

- The time to receive the list of modules: We get the system's time before the client writes the word 'MODULES' to the socket. We get the system's time again when it has finished reading from the socket. The difference is the time we print out (in milliseconds).
- The time to parse the XML string: We get the time after it has received the XML string, and when it has parsed the XML string and created all the Module objects it contained.
- The time to initiate: It is the time the client takes to load the workspace area, and the panel with the modules and the buttons, from the moment it has began running. This measurement also contains the two previous measurements, since the list of modules is received before the workspace is drawn.
- The time to create an object: When the mouse button is released, the function `createObject()` is called. We get the time after the mouse button is released, and when the function returns the object. This time is less than 1 millisecond in most cases.
- The time to save a scheme: it is the time to create the XML string of the scheme, plus the time to send the scheme to the server, starting when the OK button is pressed at the 'Save as' dialog.
- The time to execute a scheme: it is the time to create the XML string of the scheme, plus the time to send the scheme to the server, starting when the 'RUN' button is pressed. The time it takes to receive the results of the execution from the server is also measured.
- The time to load a scheme is divided into two parts:
    - The first part is the time it takes the client to read the available schemes from the server: We get the time when the client sends the word 'LOAD' to the server, and when it has read the file list from the socket.
    - The second part is the time to load the actual scheme: We get the time when the OK button is pressed at the 'Load Scheme' dialog, and when the XML string with the scheme to be loaded received and parsed and the modules have been inserted to the workspace area.

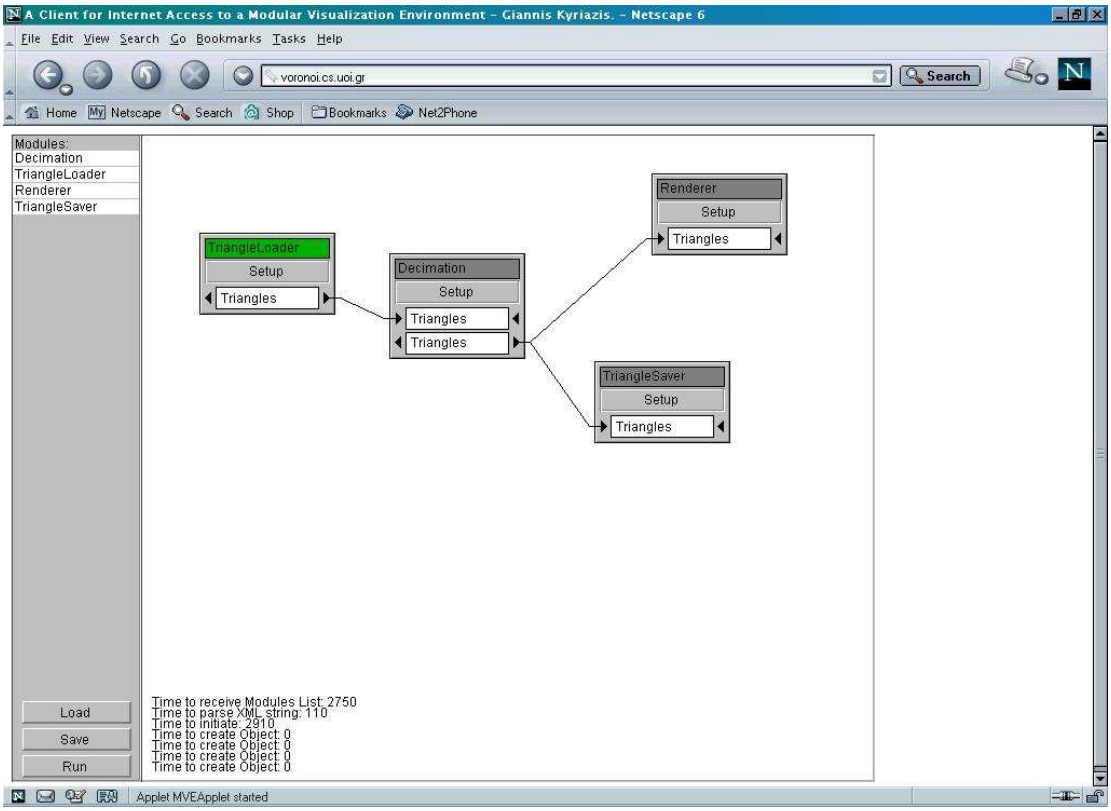


Figure 6.10: The time to insert a module into the scheme is less than a millisecond

In the Figures 6.9 and 6.10 we can see that receiving the list of available modules is the task that determines the time the client takes to load. This is the first communication between the client and the server, and it takes some time to identify to each other. Another reason for this is that the MVE_Remote cannot provide the list of modules earlier. The time to initiate, as shown in the figures, is the sum of the time to load the client, the time to receive the list of modules, and the time to parse the XML string that contains the list of modules.

In the Figure 6.10, we can also see that it takes less than one millisecond to insert a module into the scheme. That is because there is no communication required at this point, and the list of modules has already been loaded into the memory. The time to create an object for the scheme depends only on the processing capabilities of the client's machine. In situations where the client's machine was low on resources, the time to create a module for the scheme varied from 0 to 50 milliseconds.

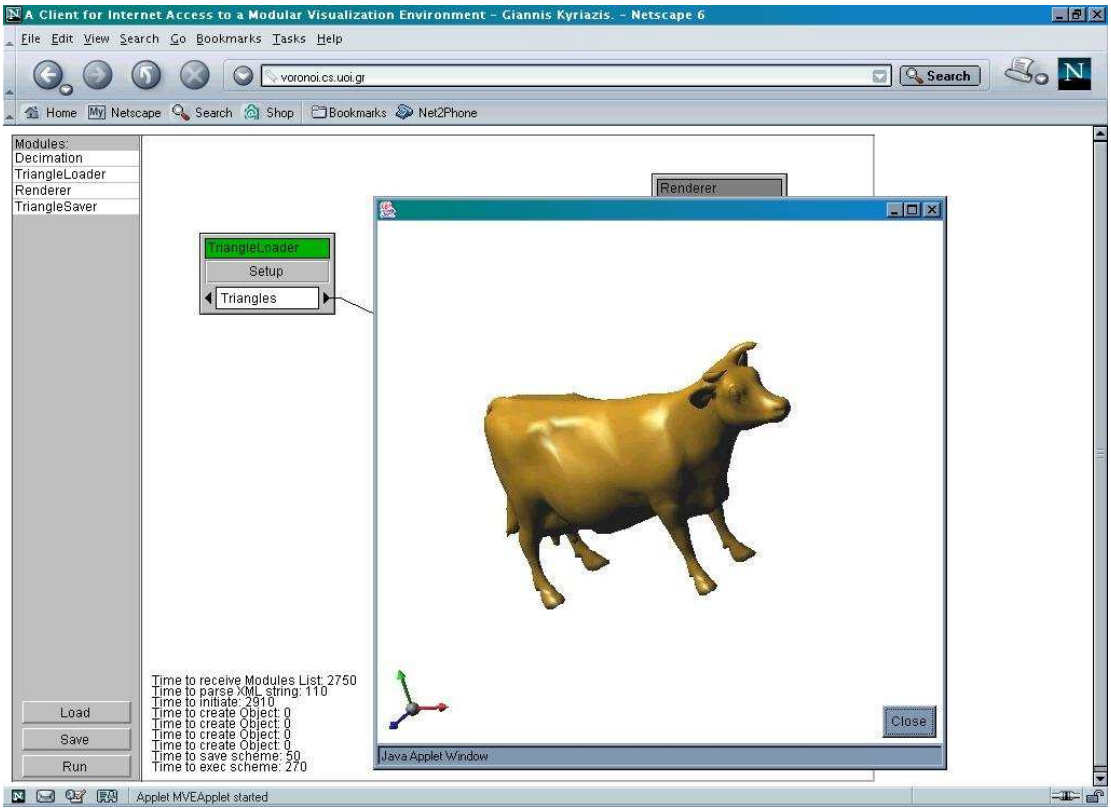


Figure 6.11: Saving and Executing a Scheme.

When it comes to saving the scheme we just created, a data exchange is required, as the scheme is saved on the server's machine. However, the amount of data to be exchanged is small, about 1kb or even less, and therefore the performance of the client will not be affected, even if the network traffic is increased. As we can see in the Figure 6.11, the scheme we created took only 50 milliseconds to be saved.

After saving the scheme, we executed it. The same XML string was transferred through the network, as when we saved the scheme. This time, however, it took 270 milliseconds to execute the scheme. The extra time was the time it took for MVE_Remote to execute the scheme that was received by the server, plus the time it took the client to load the image, which is the result of the renderer module.

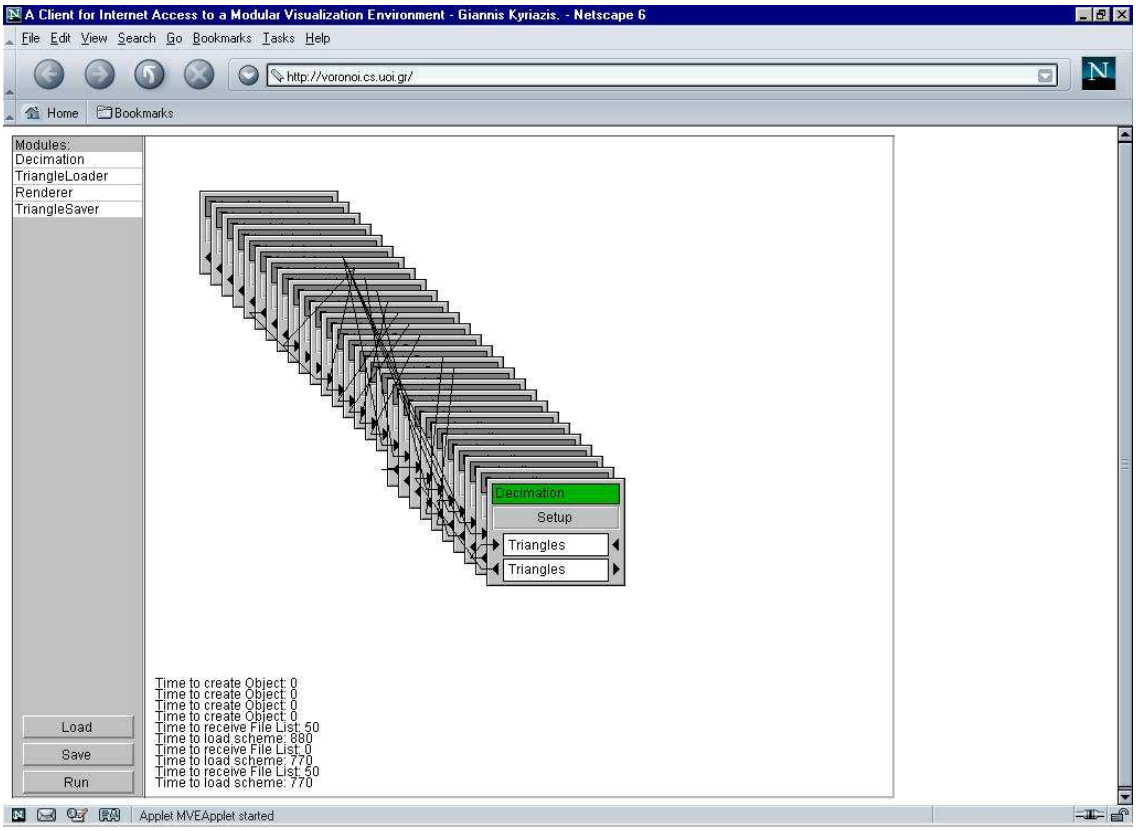


Figure 6.12: Loading a scheme with many modules.

After executing the first scheme, we loaded another scheme, that contained many modules. The scheme on the Figure 6.12 consists of 8 TriangleLoader modules, 11 Decimation modules, 5 triangleSaver modules, and 4 Renderer modules. Although the scheme is much larger than the typical scheme we used up to now, the time it took to load was not much larger. This scheme took 660 milliseconds to load. Immediately after the scheme was loaded, we pressed the load button again, and loaded the same scheme, to see if the time would be again in the same timescale. The second attempt to load the complex scheme took only 440 milliseconds! We repeated the attempt for a third time, and the time to load the large scheme was 620 milliseconds. Variations in these times are expected, as the both the client's and the server's machines have other tasks to compute as well, and the network traffic is not standard. However, as we can see, it took about half a second to load the large scheme.

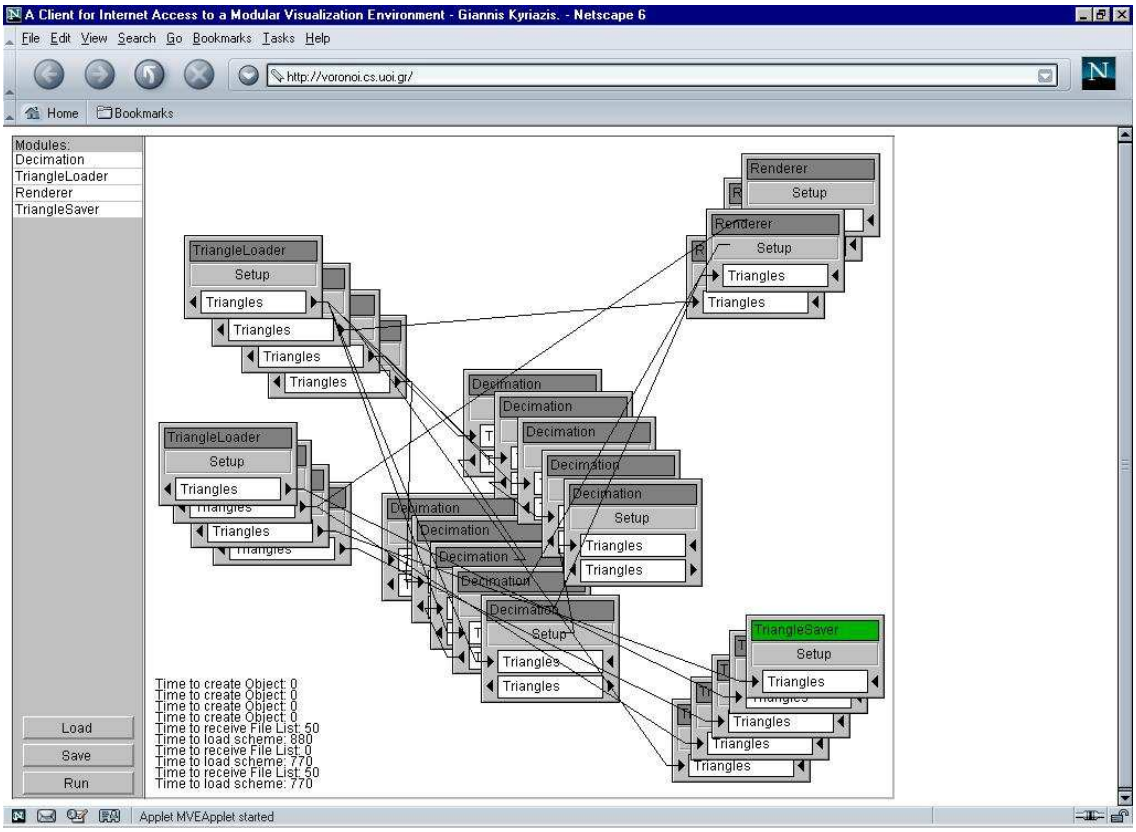


Figure 6.13: The complex scheme is executed.

After loading the complex scheme, we executed it, and it took about 2 seconds to execute. The result was 5 files that were saved on the server's machine, and 4 images that were rendered from the client.

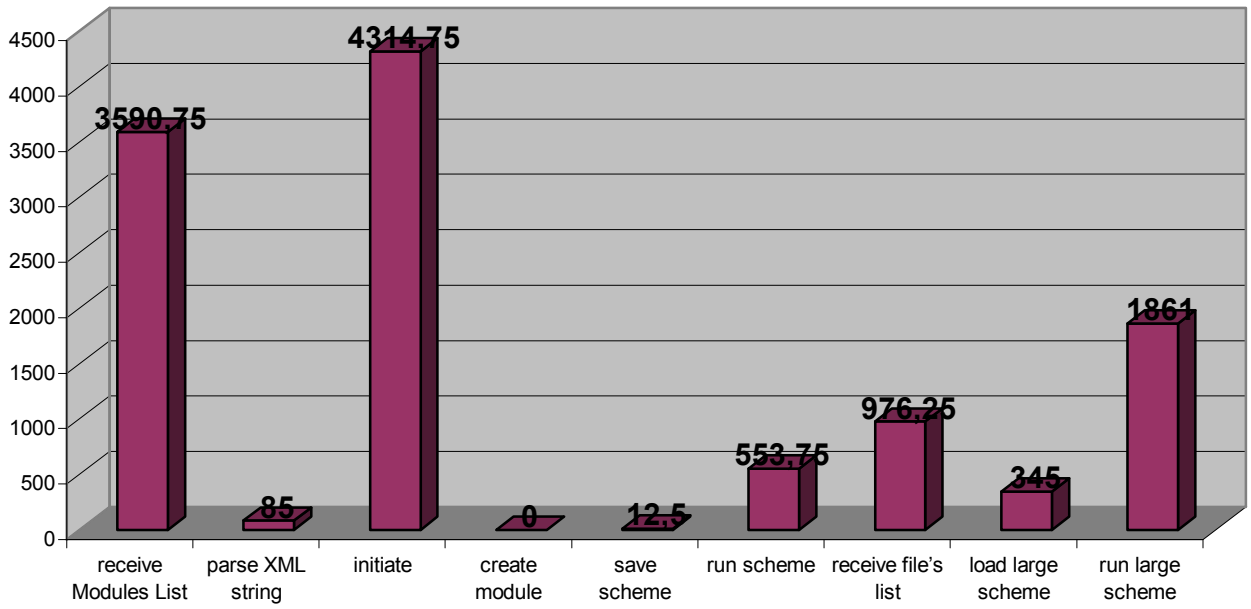
The number of modules in this scheme does not mean anything. We only wanted to create a scheme with many modules, to see how the client and the server would handle it, and how the MVE_Remote would handle it. We were surprised to see that the MVE_Remote environment crashed in some cases when we tried to execute this scheme. However, the client and the server did not have any problem with it.

The numbers given in the figures above have been measured using the Remote Machine, with a 56k modem. There was no network traffic that would reduce the client's performance, and the client's and the server's machines did not have any noticeable workload. In the Table 6.2 we can see how these times are modified as the state of the machines and the network changes.

The error value on the previous table indicates that the MVE_Remote environment crashed during the execution of the large scheme.

	Dialup Client			LAN client
Time to:	**Normal**	**Network Traffic**	**Workload on server**	
receive Modules List	2210	4030	6373	1750
parse XML string	80	100	100	60
initiate	2932	4216	8201	1910
create module	0	0	0	0
save scheme	0	50	0	0
run scheme	350	585	1010	270
receive file's list	130	370	3405	0
load large scheme	160	380	780	60
run large scheme	error	error	error	error

Table 6.1: Performance of the client under different circumstances (time in milliseconds).



Graph 6.1: The averages of the times it took the client to complete several tasks.

7. Conclusions – Future Work

As we can clearly see, our Internet Client is performing well in all cases, whether there is increased network traffic, or the machine of the client or the server is busy doing other computations as well.

So far we have achieved to design the protocol for the communication between the client and the server, and also between the server and the MVE. Through our implementation and our experiments, we proved that this protocol is suitable for our system, as it can operate successfully, based only on the data exchanged according to our protocol.

We also proved that our system performs well, under various client, server, and network configurations. It performs better than other systems used to provide remote access to Windows applications, and it performs well, even in cases where the network or the computers involved are low on resources.

We also managed to design the protocol in such a way, that the representation of the modules, the parameters and the scheme has a common simple form, making it easier to extend our system by providing support for much more and different modules, without making any changes to the system.

Our whole system and the protocol can also be easily extended to provide support for simultaneous scheme processing, so that many users can collaborate to modify the same scheme at the same time. This of course requires a slight modification of the protocol, as the server has to monitor all the modifications the users make to the common scheme that they use, and propagate the corresponding modifications to the rest of the users. This can be achieved by merging the users' actions in some way, and by saving and loading the scheme in every action each user takes. The problem with this feature, however, is that the users' actions may be conflicting in some cases, and the server has to choose the way these actions will be merged. Also, in order to propagate each action of each user would increase the communication between the clients and the server. Since the computer where the server is located is already responsible for heavy computations such as the visualization of the objects, the increase in the network traffic would be a major problem for the performance of the system. In this case, we should consider giving the clients the support to connect to

other clients directly, instead of the server, like the systems that use peer-to-peer communication. Also, if we want the users to have real-time communication with each other, we should consider transmitting only the changes on the scheme, instead of the scheme itself. This way, we would reduce even more the amount of data to be transmitted, and we could balance the increased network traffic at some point.

However, all these features require some more research, as the protocol we proposed in this master thesis becomes more or less obsolete in case we want to achieve real time collaboration. So it is left for future work.

Appendix 1

Source Code of the Client

Not included in this report

Appendix 2

Source Code of the Server

Not included in this report

References

[1] MVE - Modular Visualization Environment, Centre of Computer Graphics and Data Visualization, University of West Bohemia Czeck Republic.
The MVE and Complete Programming Documentation and User's Manual are available at **http://herakles.zcu.cz**

[2] Ioannis Fudos, An Extensible Reduced Protocol for 3D Geometric Editing, Technical Report 99-X, dept of Computer Science, University of Ioannina Greece, April 1999

[3] Amar Gupta, Sanjeev Vadhavkar, Feniosky Pena-Mora, Jason Yeung, An Integrated Framework to support Distributed CAD over the Internet, Massachusets Institute of Technology, IEEE Proceedings of the 10th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '01) 2001

[4] H. Lavana F.Brglez, R. Reese, G. Konduri A. Chandrakasan, OpenDesign: An Open User-Configurable Project Environment for Collaborative Design and Execution on the Internet, CS Dept NCSU, ECE Dept MSU, EECS Dept MIT, The proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors, 2000

[5] J. Ragusha, G. Bocheneck, Collaborative Virtual Environments, Communications of the ACM, vol 44, No12, Dec 2001

[6] R.Smith, Shared Vision, Communications of the ACM, vol 44, No12, Dec 2001

[7] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski, Executable Workflows: A Paradigm for Collaborative Design on the Internet. In Proceedings of the 34th Design Automation Conference, pages 553-558, June 1997

[8] F. Brglez, H. Lavana, Z. Fu, D. Ghosh, L. I. Moffitt, S. Nelson, J. M. Smith, and J. Zhou, Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience, In Seventh Annual Tcl/Tk Conference. USENIX, February 2000

[9] H. Lavana and F. Brglez, CollabWiseTk: A Toolkit for Rendering Stand-alone Applications Collaborative, In Seventh Annual Tcl/Tk Conference. USENIX, February 2000

[10] R. Goering, Vela Project on Collaborative Distributed Design: New Client/Server Implementations, 1999

[11] C. Rich, COLLAGEN: A Collaboration Manager for Software Interface Agents, 1998

[12] The source for Java Technology. http://java.sun.com , Complete handbook and tutorial. Source code and downloads also available

[13] Cruz-Neira, C., Sandin, D.J., DeFanti, T.A., Kenyon, R.V., and Hart, J.C. The Cave: Audio Visual Experience Automatic Virtual Environment. Comm. ACM 35, 6 (1992), 64-72

[14] R. Bowen Loftin, Design Engineering in Virtual Environments, Communications of the ACM, Vol 44, No 12, December 2001

[15] Simon Su, R. Bowen Loftin, A Shared Virtual Environment for Exploring and Designing Molecules, Communications of the ACM, Vol 44, No 12, December 2001

[16] Wing Ho Leung, Khalid Goudeaux, Sooksan Panichpapiboon, Sy-Bor Wang and Tsuhan Vhen, Networked Intelligent Collaborative Environment (NetICE), Electrical and Computer Engineering Carnegie Mellon University Pittsburgh

[17] J.Leigh, A.Johnson, C.Vasilakis, T.DeFanti, Multi-Perspective Collaborative Design in Persistent Networked Virtual Environments, Electronic Visualization Laboratory, University of Illinois at Chicago, IEEE Proceedings of the 1996 Virtual Reality Annual International Symposium, 1996

[18] S.Benford, C.Greenhalgh, T.Rodden, J.Pycock, To what extent is Cyberspace really a Space? – Collaborative Virtual Environments, Communications of the ACM, Vol 44, July 2001